TELEMATICS RESEARCH AT PRINCETON - 1987

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

CS-TR-151-88

May 1988

TELEMATICS RESEARCH AT PRINCETON – 1987

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

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

1. Introduction

In this note we briefly summarize the database and distributed computing research we performed in the year 1987. 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 9 categories: the implementation of a high availability database system, real time database systems, multicast protocols, distributed file comparison, altruistic locking, information exchange networks, data caching in information systems, process migration, 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 Available Database System

There are two important aspects to reliability: correctness and availability. We have continued our efforts to understand how very high data availability can be provided, possibly at the expense of "weakening" correctness. 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,

---

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.

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, Abbo86]. 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 have now explored a 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 [Barb87]. A node can only run transactions that update a fragment if the agent resides at the node.

Within this framework, we have discovered a number of choices. It is still possible to achieve serializable schedules, even with these highly 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 (e.g., an acyclic read access graph) and updates must be propagated to the copies in a controlled fashion [Koga87].

Another possibility is to provide *virtual serializability* [Koga87]. With this weaker type of correctness criterion, some anomalies may occur, but they are not observable by the allowed transactions. Thus, from the point of view of what transactions see, it is equivalent to strict serializabilty. With virtual serializability, the update propagation restrictions can be eliminated in some cases, yielding more up to date data.

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.

3. 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. 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)?

As a first step we have developed a family of locking-based scheduling algorithms for real time database systems [Abbo87a]. Each algorithm has three components: a policy to determine which tasks are eligible for service, a policy for assigning priorities to tasks, and a concurrency control mechanism. The eligibility policy determines whether a transaction that has already missed its deadline (or is about to) can be aborted and not performed at all. The priority policy orders the ready transactions according to their deadline, the slack time, or the arrival time. The concurrency control policy specifies the action to take when lock conflicts occur.

We have studied the performance of the scheduling algorithms via a detailed event-driven simulation [Abbo87b]. The parameters in the model include the arrival rate of transactions, the average number of objects updated, the available slack for meeting the transaction's deadline, and the possible error in the transaction's running time estimate. The performance metrics we studied were the number of missed deadlines, the transaction throughput, and the number of aborted (and restarted) transactions.

Very briefly, our results indicate that one priority policy (Earliest Deadline) and one concurrency control mechanism (Conditional Restart) are superior in most cases. They also indicate that screening for ineligible transactions can significantly reduce the number of missed deadlines (as long as the application allows this type of screening).

4. Multicast Protocols

A multicast group is a collection of processes that are the destinations of the same sequence of messages. These messages may originate at one or more source sites and the destination processes may run on one or more sites, not necessarily distinct. A multicast protocol ensures that the messages are delivered to the appropriate processes.

There are two types of properties that multicast protocols may ensure: ordering and reliability. We have studied how these properties can be *efficiently* guaranteed.

Although the two properties are related, we have initially analyzed them separately.

There are basically three ordering properties that a multicast protocol may provide. They are the following, arranged by increasing strength.

(a) *Single source ordering.* If messages m_1 and m_2 originate at the same source site, and if they are addressed to the same multicast group, then all destination processes get them in the same relative order.

(b) *Multiple source ordering.* If messages m_1 and m_2 are addressed to the same multicast group, then all destination processes get them in the same relative order (even if they come from different sources).

(c) *Multiple group ordering.* If messages m_1 and m_2 are delivered to two processes, they are delivered in the same relative order (even if they come from different sources and are addressed to different but overlapping multicast groups).

We have developed a new multicast protocol that guarantees these three properties [Spau88]. The basic idea is to organize the nodes into a logical tree structure. Messages flow down the tree and are ordered as they move along. The number of messages that must be transmitted is much smaller than in other fully distributed solutions. In many cases, our broadcast tree will have small depth and the multicast delay may also be smaller than in conventional solutions. The disadvantage with our approach is that the tree structure must be set up, and hence it is not advisable for applications where the multicast groups are rapidly changing. In [Spau88] we also argue that the reliability that can be provided by our approach is comparable to that of other strategies.

A *reliable* multicast protocol must ensure that all destinations eventually receive all messages. In order to understand the cost of providing reliability, in another study [Garc88] we have focused on reliability in point-to-point networks. (We assume a single multicast group and ignore ordering problems.)

A simple way to perform a reliable multicast would be to send a separately addressed copy to every destination and to repeat this process until an acknowledgement was received. Not only is this solution expensive in terms of the number of messages, but it also fails to propagate the messages efficiently. Consider for example a situation where the source gets disconnected from the network after delivering a message only to a portion of the destinations in the multicast group. With the simple algorithm, the remaining hosts will have to wait until the source is reconnected. However, a smarter algorithm would make the destinations with a copy propagate the message to the remaining destinations.

Such smarter algorithms have been proposed, but they all assume that the servers within the network are programmable and can run the multicast protocol. However, in reality, most networks do not allow users to program their internal servers, so we have developed an efficient reliable multicast protocol that only runs on the user machines (called hosts) [Garc88]. This algorithm also creates a multicast tree structure. When failures occur, the structure is dynamically adjusted. For this adjustment, nodes must periodically probe neighbors (and parents on the tree) to detect if more up to date information is available. The more frequent the probes and the adjustments, the more effective the multicast becomes. However, the overhead of the probes and adjustments also grows. Our algorithm uses heuristics to select the right dynamic

reconfiguration strategy.

5. Distributed File Comparisons

Files are replicated in a distributed system in order to improve reliability and performance. However, due to human errors or hardware failures copies may diverge. It then becomes necessary to compare the remotely located files and identify the differences. A simple approach would be to transmit the entire file and do a compare at a single site. However, more sophisticated approaches are required in cases where the files are large and the cost of identifying the differences must be kept low. Such a case arrises, for example, in a triple modular redundant (TMR) database system that has been built at Princeton [Abbo87c].

We have developed a file comparison mechanism that can identify up to two differing file pages with a single message transmission [Barb]. The basic idea is that each site maintains a collection of *signatures* for the database. To compare two remotely located files, one site sends its signatures to the other. The second site compares the two sets of signatures. If there are two or less pages that differ, then the second site can immediately determine what these pages are. If there are three or more page differences, the algorithm detects this are yields a set of pages that is a superset of the pages that differ. Since the algorithm only compares probabilistic signatures, there is always a small probability that the result will be incorrect (e.g., there may be undetected differences). By making the signatures lager, the probability of error can be made arbitrarily small.

Previous solutions to this problem either required many more messages to pinpoint a difference, or could only identify a single differing page. Our approach can pinpoint one or two differences with a single message. Although the improvement form single to double identification may seem small, there are cases where it is important to identify two differences. (For example, a disk failure may easily affect two contiguous pages.) Furthermore, the extra bookkeeping cost of our approach is reasonable. In particular, if the file contains 2^m pages, we must store and update $m(m+1) + 1$ signatures, while the one identification strategy requires only $m + 1$. For files with less than, say, a million pages, the extra cost may be worth the ability to cover more failures.

6. Altruistic Locking

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, LLTs 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. One option is to relax the requirement that the LLT be executed as an atomic action. This leads to the concept of *saga* that we studied earlier [Abbo87c, Garc87]. More recently, we have explored another option, *altruistic locking* [Sale87].

With altruistic locking, a LLT makes available to other transactions, before the end of the LLT, objects it has locked. The LLT must not access again the objects it has released. Furthermore, other transactions that do access this released data must not process data that will be touched by the LLT in the future. In other words, we can view the LLT as a ship that is leaving a wake of released data. The LLT never returns to its wake, and transaction must not straddle the wake and data outside the wake.

The altruistic locking rules are simple extensions to conventional locking rules that force these properties. Serializable schedules are produced, in spite of the early releases. The early lock releases do not benefit the LLT directly, but can significantly reduce the delays encountered by other shorter transactions, hence the name altruistic locking.

In order to release locks early, a LLT must know its reference pattern, i.e., must know that the released objects will not be needed again. We believe that this knowledge may be common in LLT that must scan large portions of the database. In addition, some LLTs may know that they will never access some data. This data may be *marked* by the LLT. A short transaction running in the wake of the LLT may access marked data, in essence moving it into the wake on behalf of the LLT. The locking rules for marked data are given in [Sale87]. In that report, we also present a simple performance analysis that helps identify the situations where altruistic locking may be helpful.

7. Information Exchange Networks

We are studying a computing environment likely to evolve within the next several years, one in which large numbers (thousands or millions) of independent systems can communicate over an ISDN-type network. The dominant characteristics of this environment are its large scale, extreme heterogeneity of the participating systems (nodes), and the necessary autonomy of those nodes. Autonomy in particular is a requirement not often considered in distributed systems research, and we are exploring its many implications. For example, it appears impossible to design distributed applications from the top down, as we are accustomed to doing; they must evolve from the bottom up, under the decentralized control of the individual nodes. Participation in any application must be completely voluntary and revokable by the participant at any time.

A natural application for this environment is information retrieval. Thousands of publicly accessible computerized information services already exist, and with personal computers (PCs) within the reach of the average person, individuals increasingly will store their important information in computer files and databases, as most businesses

do now. It will be desirable to exchange one's information with others, and to add to one's personal database items received directly from other machines. Specifically, we envision the personal computer as a gateway to a world of information sources: via a customized interface, a user (querier) may ask about any topic, his PC will locate sources capable of providing information on that topic, query those sources, and collect the responses. We are developing an "architecture" for large-scale information retrieval, and have divided the problem into two parts: external indexing and query translation.

External Indexing. The external indexing problem is that of discovering what nodes exist, and what sort of information each can provide. Assuming a very large network of changing composition, a complete, consistent, universally accessible index to network contents is impossible. The index used will then be a distributed, dynamic, and often inconsistent one. Our goal is for each node to store locally as much of this index as it wants and/or needs, and to be able to query remote portions of the index when the local portion is insufficient or out-of-date. We call each node's version of the index its *directory*. The directory is a list of node addresses together with summaries of the information each can provide. These summaries, which we call *node descriptors* (NDs), are supplied by the source nodes themselves. (Note that any node may act as a source, a querier, or both). Nodes may acquire new NDs for their directories by asking neighboring nodes (known sources) for copies of their directories, by asking source nodes for their own NDs, and by receiving unsolicited "announcements" of NDs from information sources. An ND is similar to a database schema, as it must describe both the content and, to some degree, the structure of the available data. Much of our present work involves devising a simple yet expressive ND format, one in which NDs can be generated automatically from schemas of databases of various types. The primary criterion for choosing a source to answer a query is the similarity of its ND to the query in question, but additional directory information about the cost, timeliness, trustworthiness, availability, etc. of a source's data may also come into play.

Query translation. The ability to query heterogeneous sources requires mechanisms for translating user queries into the internal query language of many data retrieval systems. Sources may be true DBMSs (e.g. relational DBMSs), term-based retrieval systems, or even simple file systems. There are two possibilities for distributing the translation function: either NDs will include a description of sources' query formats, and the querier's software will translate queries to them, or all nodes will recognize a universal query transmission language, to which queriers will translate their queries and which each source will translate into its internal query language. We have investigated the second possibility. We have designed a very simple query format whose main component is a set of terms (essentially natural-language words or word stems) with attached weights. Such queries are easily generated from natural language queries, and in [Simp88] we gave algorithms for translating received queries into queries on two types of information retrieval system: term-vector systems and Boolean systems. Present research concerns their translation onto relational DBMSs. Since queries and NDs should be of compatible format for the purpose of locating sources to answer a particular query, the development of formats for queries and for database schemas is proceeding in parallel.

8. 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.

However, although in principle caching may offer a number of benefits, it also has an associated cost. 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 [Alon88a] 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.

9. Process Migration

Process migration is the capability to move a process that is running on a certain machine to another, without interrupting its execution. This is a useful feature to have, as it can be used in various ways, ranging from system applications such as load balancing and process checkpointing, to applications for individual users, such as moving a process from a machine that is about to go down to another. However, implementing process migration in an operating system is a non-trivial task. One must be able to keep track of all the system resources that a process is using and to reallocate them to the migrated process on another machine. For example, one must know what files a process is using and have the ability to reopen these files on behalf of the migrated process on another machine, keeping track of the current offset within the file. If files are local to the machine that owns the device on which they are stored, this can prove to be very hard, if not impossible, as in many cases the file system cannot access the files of a remote machine directly.

We have implemented a process migration mechanism on Sun 2 workstations under version 3.0 of the Sun operating system. (The details of our implementation have been presented in [Alon88b].) In our system, processes that do not communicate with other processes and that do not take actions that depend on knowledge of the execution environment (such as the process id), can be moved from one machine to another while running, in a transparent way. Furthermore, the overhead of moving a job is relatively small (a few seconds in the worst case for most processes).

In our implementation, process migration is achieved by signaling a process to stop, saving all the kernel and memory information that is necessary to restart the process, and then, by using this information, restarting the process on the new machine. This new functionality required minor kernel modifications as well as the creation of a new signal and a new system call. The new signal stops a process and dumps its core, and it also includes in the dump the information the kernel keeps about open files (such as the value of the file offset). The added system call restarts a process using the information dumped by our signal.

Currently, we are examining in detail the possible benefits of process migration in a load balancing scheme. By a load balancing scheme we mean a mechanism that tries to execute tasks in the least loaded processors of a system. If the scheduling decision can only be made when the job first starts up the load balancing scheme is said to be of the initial placement type. These schemes have the potential disadvantage that they may not recover from errors (i.e., if too many jobs end up at one site the jobs may not be moved elsewhere). If a process migration tool is available the combined system may exhibit much better behavior.

However, the extent of the improvement is not clear. There are some theoretical studies that indicate that the potential benefit of process migration (when compared to an initial placement policy) is relatively small. We are currently carrying out experiments to determine the conditions (if any) under which process migration provides a sizable performance gain.

10. 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.

In the past, we have reported on our work developing a load balancing implementation that runs on a network of SUN workstations [Alon87]. Although our previous research was oriented towards policies that tried to evenly spread the system load throughout the network, our present direction 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 studying load sharing in networks where individual users have a large amount of autonomy over their machines. In such environments it is not appropriate to evenly share the load, but users may be willing to help out as long as their own performance is not disturbed greatly. The algorithm we have developed (*High-Low*) involves the use of two thresholds: a low watermark and a high watermark. If system load at any one site is below the low watermark that system is considered to be lightly-loaded, and thus will be willing to process remote jobs. If the load is between the two thresholds, the processor is in its normal state, and will neither accept remote jobs, nor request help from other nodes. When a machine's load increases past the high watermark that processor is allowed to try to send some of its local jobs to remote hosts.

We have implemented this simple scheme and it has proved very effective. By varying the relative values of the two watermarks we can obtain a number of different policies spanning the spectrum from least autonomy and greatest sharing (forcing all processors to have the same load) to complete autonomy and no sharing (each processor executes only local jobs). The results of this work appears in [Alon88c].

We are presently conducting experiments to determine how High-Low compares with other techniques for load sharing that also respect processor autonomy (i.e., running remotely only on idle processors, or executing migrated jobs at a lower priority

than local jobs).

11. References

[Abbo86] R. Abbott et al, "Distributed Computing Research at Princeton – 1985," Technical Report CS-29, Department of Computer Science, Princeton University, 1985.

[Abbo87a] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions," submitted for publication, December 1987.

[Abbo87b] R. Abbott, H. Garcia-Molina, "Scheduling Real Time Transactions: A Performance Evaluation," submitted for publication, February 1988.

[Abbo87c] R. Abbott et al, "Distributed Computing Research at Princeton – 1986," Technical Report CS-TR-073-87, Department of Computer Science, Princeton University, January 1987.

[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.

[Alon87] Rafael Alonso, "An Experimental Evaluation of Load Balancing Strategies," *Proceedings of Miami Technicon '87*, October 28-30, 1987, Miami, Florida.

[Alon88a] Rafael Alonso, Daniel Barbara, Hector Garcia-Molina, and Soraya Abad, "Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems," *Proceedings of the 1988 International Conference on Extending Data Base Technology*, March 14-18, 1988, Venice, Italy.

[Alon88b] Rafael Alonso and Kriton Kyrimis, "A Process Migration Implementation for a UNIX System," *Proceedings of the Winter 1988 USENIX Conference*, February 9-12, 1988, Dallas, Texas. An extended version appears in Technical Report TR 092-87, Department of Computer Science, Princeton University, May 1987.

[Alon88c] Rafael Alonso and Luis Cova, "Sharing Jobs Among Independently Owned Processors," *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 13-17, 1988, San Jose, California.

[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*, March 1987, pp. 177-187.

[Barb88] D. Barbara, H. Garcia-Molina, B. Feijoo, "Exploiting Symmetries for Low Cost Comparison of File Copies," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, San Jose, June 1988, to appear.

[Garc87] H. Garcia-Molina, K. Salem, "Sagas," *Proceedings of the 1987 SIGMOD International Conference on Management of Data*, May 1987, pp. 249-259.

[Garc88] H. Garcia-Molina, B. Kogan, N. Lynch, "Reliable Broadcast in Networks With Nonprogrammable Servers," *Proceedings of the IEEE International*

Conference on Distributed Computing Systems, San Jose, June 1988, to appear.

[Koga87] B. Kogan, H. Garcia-Molina, "Update Propagation in Bakunin Data Networks," *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, August 1987, pp. 13-26.

[Sale87] K. Salem, H. Garcia-Molina, R. Alonso, "Altruistic Locking: A Strategy for Coping with Long Lived Transactions," *Proceedings of the ACM-IEEE Second International Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1987.

[Simp88] Patricia Simpson, "Query Processing in a Heterogeneous Retrieval Network," *Proc. Eleventh Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, June 1988, to appear.

[Spau88] H. Garcia-Molina, A. Spauster, "Message Ordering in a Multicast Environment," submitted for publication, January 1988.