Achieving Transaction Scaleup on Unix

Marie-Anne Neimat and Donovan A. Schneider

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
latname@hpl.hp.com

Abstract

Constructing scalable high-performance applications on commodity hardware running the Unix operating system is a problem that must be addressed in several application domains. We relate our experience in achieving transaction scaleup on Unix for a high-performance OLTP system intended for Service Control Points (SCPs) in a telephone switching network. We view the requirements of SCPs as prototypical requirements of a class of applications that cannot be properly handled by today's commercial DBMSs. In addition to high throughput and low response time, SCPs require transaction scaleup on standard hardware and software architectures. Using a main-memory DBMS to obtain high throughput, we focus on the problem of achieving transaction scaleup on a cluster of workstations running Unix while constrained by the low response time requirement of the SCP application. The use of a main-memory DBMS causes the throughput and response time to be much more sensitive to the cost of messages and more susceptible to the formation of convoys than they would have been with a disk-based DBMS. We relate the various experiments and discoveries of what goes on underneath the covers in Unix that led to our choice of architecture, inter-process communication mechanisms and mode of running the system to achieve transaction scaleup under the constraint of low response time. This experience should provide valuable information to anyone trying to build a scalable high-performance application on Unix.

1 Introduction

The telecommunications industry has been undergoing some fundamental changes in the last few years [19]. Many of these changes have been driven by the need to easily deploy new services. In the past, installing a new service meant reprogramming a large number of expensive special-purpose switches constructed with specialized hardware, operating systems, and applications software. The trend now is to offload much of the specialized services and data lookup of the switches to Service Control Points (SCPs). These are general-purpose computers that are relatively inexpensive when compared to the cost of switches, easy to program, and easy to customize by telecommunications companies and some of their customers. At the heart of the SCP is a giant database with stringent data access and throughput requirements. The operating system and the interfaces to software running on these systems must be "standard". This insistence on "open systems" is no surprise as it is typical of today's trend in all application domains. The data management requirements of SCPs include very high throughput, low response time and scalability on open systems.

The motivation for the work described in this paper was to meet these data management requirements on Unix. The relevance of these requirements should not be confined to SCPs as they are representative of a class of OLTP applications with data management demands that exceed the capabilities of today's commercial DBMSs. Home Location Registers, as found in mobile telephone networks, have similar requirements albeit for write-intensive queries as opposed to the read-intensive queries of SCPs. Financial applications have similar requirements.

As we explain later, the combined requirements of SCPs are such that they can only be met by a main-memory DBMS. We used Smallbase [11], a main-memory DBMS developed at HP Labs, to obtain the desired throughput rate and transaction response time on a single-node system. We then focused on the scalability requirement of the SCP application. Motivated by the necessity of using open systems, we addressed the problem of achieving transaction scaleup on Unix under the constraint of low response time.

We evaluated a distributed architecture based on Smallbase using a benchmark for which we expected transaction scaleup, i.e., for which we expected the number of Transactions Per Second (TPS) to increase linearly with the number of processors used [6]. Our main concern was whether we could maintain the constraint of low response time in a distributed system, and whether we could indeed achieve transaction scaleup. Since we were using a main-memory DBMS, we knew that the system would be very sensitive to the cost of network messages [16]. Our initial experiments on Unix did not scale, and the constraints on response time were not always met. It took numerous experiments and investigations into the implications of various Unix commands to finally discover the reasons why transaction scaleup and low response time were not achieved and how to achieve them. The fact that the system was based on a main-memory DBMS made it much much more susceptible to the formation of convoys than it would have been with a disk-based system.

Extensive research has taken place in the design, benchmarking and tuning of main memory DBMSs, a subset of which may be found in [7, 8, 13, 15, 16]. This work has had a major influence on the design of Smallbase. On the other hand, research in parallel/distributed main memory DBMSs has been substantially more limited. Prisma [21] is a main memory parallel DBMS where the focus of the research has been to obtain linear speedup on "decision support" queries. In the TPK system [16], Li and Naughton combine large main memories and multi-processors to obtain high transaction rates. The focus of their work is on using parallelism to speed up the most time-consuming components of a transaction and on the judicious grouping of work to optimize throughput. The work we report in this paper is unique in that its goal is to achieve transaction scaleup on Unix, and its contribution is to report on the architectural design of a distributed OLTP system based on a main-memory DBMS, on the choice of inter-process communication, and on the mode of running the system to achieve the desired goal.

The remainder of the paper is organized as follows. Section 2 describes SCPs, their functionality and requirements. Section 3 describes the benchmark we used. Section 4 describes the architecture of the distributed system and the justification for such an architecture. Section 5 reports the results of the experiments we ran. Section 6 concludes with the lessons we have learned.

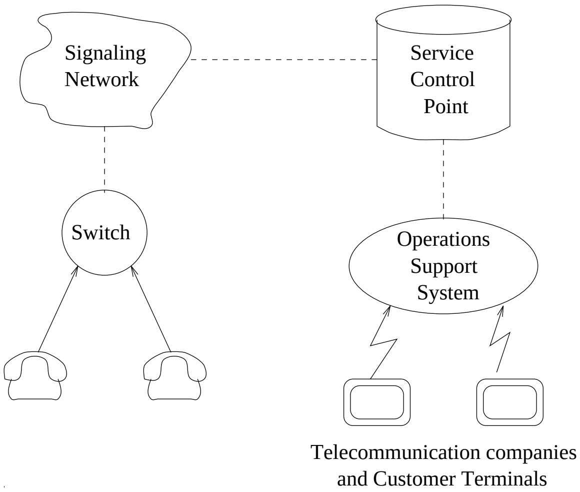

Figure 1: *Intelligent Network Architecture*

2 Service Control Points

Telephone switches evolved from manually operated switches to switches that were programmable by the specialized switch manufacturers [19]. Although this represented a major improvement over the manually operated switches of the past, it still meant that new services were hard to deploy as they required modifications by experts to non-standard systems. It also meant that to make a new service available to a specific geographic area, new software had to be downloaded in the switches servicing that area. This is an expensive and time-consuming process as the number of switches tends to be very large (over 15,000 in the US alone.)

To solve this problem as well as the interoperability challenge that followed the divestiture of the Bell Operating Companies from AT&T, the new regional companies defined the architecture of the Intelligent Network (IN), a network architecture with capabilities for advanced services. A high-level picture of the IN Architecture is shown in Figure 1. When a call is received by the switching system, if the call requires special processing such as 800 number lookup or the processing of a credit card call, a request is sent to the SCP where it is serviced. The SCP executes the service logic used to control the call processing of the switch. It interacts with a database that contains information about subscribers, the services to which they are subscribed, the service logic, operational and configuration data, and vendor-specific configuration information. The response is sent back to the switching system to complete the call. The role of the operations support system is to permit the administration of the database managed by the SCP and the introduction and modification of new services.

The services provided by the SCP could have been handled in the switch. However, by taking them out of the switch and into a centralized database, new services can be made available to a large number of switches (and hence a large geographical area) uniformly, simultaneously and inexpensively. Hence, the main contribution of the IN was to introduce the concept of a centralized database that could be used to deploy new services easily and uniformly.

The second phase of the IN, named the Advanced Intelligent Network (AIN), focuses on the easy development and customization of new services, known as *service creation*. This permits a non-programmer to introduce new services by gluing together system-provided, software construction blocks. The service creation environment must interact with the SCP database to install new services. The goal is to make the creation of new services easy not only for the telecommunications companies, but also for their customers.

The SCP is then at the heart of the IN and AIN architectures. Telecommunications companies are insisting that SCP platforms be general-purpose computers running open systems. The operations executed by the SCP are dominated by the interaction with its database. Hence, an efficient DBMS platform for SCPs is essential to the successful support of the IN and AIN.

We examine now the DBMS requirements of the SCP [1]. Most of the performance requirements specified by Bellcore are in terms of overall performance of the SCP, and not specifically as it pertains to the DBMS component of the SCP. Hence, the numbers reported here as DBMS requirements are extrapolations from these more general SCP requirements to guarantee that the general requirements are met. The SCP DBMS requirements are [17] 2,000 TPS, single figure milliseconds response time, a database size that varies from 40 megabytes to tens of gigabytes, no more than 3 minute down time per year, and the restriction that no more than 1 transaction in 1,000,000 be lost. Scalability in throughput is expected to accommodate increases in the number of supported subscribers. The size of the database and the number of subscribers are strongly correlated.

The SCP transactions are simple, read-intensive, OLTP-type transactions. Update transactions occur relatively infrequently. They reflect events such as the installation of a new service, or the registry of a new telephone number mapping in the case of a personal locator service which allows a subscriber to use a single phone number independently of where he is.

Given the type of transactions of the SCP, the 2,000 TPS throughput could, in principle, be met by today's commercial disk-based DBMSs on a multi-computer system. This is however an extravagant expense for small databases. To meet the 2,000 TPS requirement on a single-computer system, the number of instructions required to execute a transaction must be reduced considerably over that of conventional DBMSs. Main-memory DBMSs [7, 13, 15, 16] have been shown to reduce the instruction path of a transaction enough to make that throughput rate achievable. In fact, Smallbase [11] can easily meet this throughput rate.

To meet the constraint on response time, it is essential that all I/O operations be taken out of a transaction path. The use of a main-memory DBMS certainly helps, but it does not eliminate the crucial log write to disk on commit that is needed to guarantee the durability of transactions. Many techniques can be used to eliminate the disk write from the transaction path. Keeping the log in safe RAM and spooling it to disk in the background is one possibility [5]. Similar to the notion of *levels of safety* [14] used in the context of disaster recovery, a notion of levels of durability can also be devised. For example, one could define level 1-durable to mean that the log is not posted to disk in the path of the transaction, level 2-durable to mean that the log is posted to the main memory of a hot standby, and level

3-durable to mean the conventional definition of posting the log to disk. Of course with relaxed notions of durability, one could lose some transactions. The degree of tolerance for such losses and consequently the level of durability is dependent on the application. We thus assume that a main-memory DBMS with appropriate hardware support and/or more flexible notions of durability can meet the throughput and response time requirements of the SCP application on a single-node system.

Although not the topic of this paper, high availability can be easily achieved with a hot standby using log shipping as is done in Tandem's NonStop SQL [3]. When coupled with scalability, one should not double the number of nodes so that each node has a dedicated hot standby. Instead, each node could act as the primary for one partition of the database and as the hot standby for another partition. The assignment of database partitions to primaries and hot standbys could be done using chained declustering [12], with the partitions residing in main memory instead of on disk. This technique of assigning partitions to nodes has been used for many years by Tandem's customers [4] and has been proposed in [2]. We do not further discuss the issue of high availability in this paper.

We concentrate, instead, on achieving transactional scaleup for a main-memory DBMS on commodity hardware running the Unix operating system.

3 Benchmark

To experiment with different architecture choices and measure the scalability of the system, we chose to use the TPC-B OLTP benchmark as our sample application [9]. The TPC-B benchmark does not truly model an SCP because SPCs are read intensive while the TPC-B benchmark is write intensive. Nevertheless, we chose to use it for several reasons. First, given the popularity of the TPC-B benchmark, the performance numbers we obtain have a better-understood context than if we devise our own benchmark. Second, if we obtain favorable results with the TPC-B benchmark, the results will also be favorable to the SCP application. Finally, although focused on the SCP application, other applications like Home Location Registers are write-intensive, and we wanted the results we obtain to be meaningful in a more general context.

The TPC-B OLTP benchmark simulates a hypothetical bank with one or more branches, tellers and many customer accounts. The database maintains current balances for each account, branch and teller, plus a history of recent transactions.

The database consists of four separate files/tables: Account, Teller, Branch and History as shown in Figure 2. There is a one-to-many relationship between BranchID and Account-BranchId and between BranchID and TellerBranchId. Account, Teller, and Branch records must contain at least 100 bytes. For each TPS that a system claims to perform, there must be at least 100,000 Account records, 10 Teller records and 1 Branch record. Hence a database size must grow by roughly 10MB for each TPS. The benchmark also specifies that 90% of all transactions must complete in less that 2 seconds.

A transaction consists of 6 SQL statements: 3 updates, 1 select, 1 insert, and 1 commit as shown below:

1. ```
   UPDATE Account
   SET AccountBalance = AccountBalance + :amount
   ```
   ```

Account:	< AccountID, AccountBalance, AccountBranchId, AccountName >
Teller:	< TellerID, TellerBalance, TellerBranchId, TellerName >
Branch:	< BranchID, BranchBalance, BranchName >
History:	< AccountID, TellerID, BranchID, amount, time_stamp >

Figure 2: *TPC-B Schema*

```
   WHERE AccountID = :AccountID;

2. SELECT AccountBalance INTO :AccountBalance
   FROM Account WHERE AccountID = :AccountID;

3. UPDATE Teller
   SET TellerBalance = TellerBalance + :amount
   WHERE TellerID = :TellerID;

4. UPDATE Branch
   SET BranchBalance = BranchBalance + :amount
   WHERE BranchID = :BranchID;

5. INSERT INTO history(AccountID, TellerID, BranchID, amount, time_stamp)
   VALUES( :TellerID, :BranchID, :AccountID, :amount, curtime);

6. COMMIT WORK;
```

One can observe that the input parameters to a transaction are *AccountID*, *TellerID*, *BranchID*, and *amount*. Given a *BranchID*, the benchmark specifies that the *TellerID* generated must belong to the *BranchID*. On the other hand, the *AccountID* should belong to the *BranchID* with an 85% probability and to a different branch with a 15% probability.

For our prototype, we horizontally partition the 4 tables over a cluster of nodes. We interpret the benchmark specification in the following manner: for each branch residing at a given node, the 10 tellers and 100,000 accounts associated with it will reside at the same node; for a given transaction, the *BranchID* and *TellerID* will belong to the node on which the transaction is generated, and the *AccountID* will belong to the local node with an 85% probability and to a different node with a 15% probability[1].

The TPC-B benchmark queries do not involve joins and each statement can be executed in its entirety on a single, possibly remote, node. The *AccountID*, *TellerID*, and *BranchID* are known at the beginning of the transaction, and the only dependency between SQL statements is between statements **(1)** and **(2)** where statement **(1)** must be executed before statement **(2)**. If the *AccountID* belongs to the local node, the entire transaction can be handled locally. If the *AccountID* belongs to another node, the SQL statements **(1)**, and **(2)** will have to be executed on a remote node, while the SQL statements **(3)**, **(4)** and **(5)** will be executed on the local node.

We omit the commit statement **(6)** from the benchmark because Smallbase does not yet support the ACID properties of transactions. It will support them in the future, but we

[1]This is a simplification; a remote branch may be stored locally. 15% remote transactions is the worst case.

recall that different notions of durability will have to be implemented to meet the response time constraint of the SCP application. For TPC-B and SCP-type applications where there is very little contention on the data, the lack of transaction management affects the actual throughput and response time numbers, but it does not affect the scalability of the system. In other words, if throughput scales linearly without transaction management, it should also scale linearly with transaction management, albeit with different TPS numbers. Given that our goal was transaction scaleup, the actual throughput numbers were not important to us in and of themselves, provided they were high enough to meet our throughput goal even in the presence of transaction management. As for response time, if we assume that I/O has been taken out of the transaction path, our only concern should be for distributed transactions where 2-phase commit would normally have to take place. We ensure in our experiments that there is enough slack in the response time of distributed transactions that they can pay for the additional cost of 2-phase commit while still remaining within the constraints on their response time.

4 Prototype Architecture

The architecture of the system consists of copies of Smallbase residing on multiple nodes of a computer cluster. The data for the TPC-B application is horizontally distributed amongst the nodes in the cluster using value-range partitioning. Each copy of Smallbase assumes that it owns the entire database and is unaware of the other nodes. Knowledge of data distribution is implemented outside Smallbase. Client processes generate the transactions and submit them for execution. We chose to run the client processes on the same nodes of the cluster, i.e., there is no dedicated processor to run the clients. Inter-process communication (IPC) is needed for distributed and for local transactions. In the remainder of this section, we discuss our choice of IPC and the overall architecture of the system including the decisions that led to it.

4.1 Inter-Process Communication

For remote IPC, we needed a communication protocol that guarantees the reliable delivery of messages. Two user-level communication protocols built on top of the Internet Protocol (IP) are the Internet Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). IP is used to transmit blocks of data, also known as datagrams, between two nodes connected by a packet-switched computer communication network. IP transmits datagrams through the local network protocol, and handles services such as the fragmentation and reassembly of long datagrams if necessary. IP is not a reliable communication mechanism. There are no acknowledgements or error control for data.

TCP, built on top of IP, provides ordered, reliable delivery of streams of data between pairs of processes in interconnected computers. It supports two-way transmission of data between the processes. It requires the setting up of a connection between the two processes prior to their exchanging messages.

UDP, also built on top of IP, provides support for sending datagrams from one application program to another with as few details of the protocol as possible. UDP does not enhance

IP with reliable delivery of messages or with error control of data. It does not require setting up a connection between the communicating processes.

For our data-centric application, the reliable delivery of messages is essential. Since neither the TPC-B nor the SCP application requires message streams, reliable datagram messages would have been the ideal because they do not require a prewired connection between the communicating processes. We were however forced to use TCP because of its reliable delivery of messages.

Both stream connections and datagram connections are available through Unix sockets, an application program interface to the communication protocols. When used to communicate between processes on different nodes on a network, a socket stream connection will use the TCP protocol. That is the mechanism we used for remote communication.

As for the choice of IPC within a single node, two overriding concerns dominated our decision making. First, we wanted to choose an IPC mechanism that executes as few instructions as possible. Second, we did not want to use polling for process synchronization so that the CPU would be always executing useful work. We should point out that we were willing to adopt a different IPC mechanism for local communication than from remote communication to guarantee the best performance on a single-node configuration. This was essential as the SCP can often be configured as a single-node system (ignoring the presence of a hot standby).

On a single-node, the mechanisms available for one process to send data to another are: sockets, pipes, named pipes (FIFOs), message queues, and shared memory coupled with semaphores. Shared memory is the fastest and most flexible way for one process to pass data to another as it requires no data copying. The sender writes data in shared memory where the receiver can read it. However, it does need to be coupled with a synchronization mechanism so that reading and writing operations are atomic and properly interleaved. Semaphores are typically coupled with shared memory. They provide the desired blocking for synchronization as well as process scheduling. Of these five mechanisms, shared memory coupled with semaphores is the fastest [20] as it requires no data copying into kernel buffers.

4.2 Description of architecture

As justified earlier, we chose TCP/IP for two processes to communicate across the network. TCP/IP requires setting up a connection prior to sending a message. The building and tearing of such connections is too expensive to execute on the fly. Hence, for a high-performance application, any two processes that are intended to communicate via TCP/IP should set up a connection that would last while the application is running. TCP/IP connections consume main-memory kernel buffers and file descriptors. This obviously presents a scalability problem as it implies that every pair of processes that reside on different nodes and that need to communicate must have this connection set up. It is thus important to limit the number of processes that must directly communicate via TCP/IP. To this end, we chose to dedicate one process on each node for sending network messages and one process on each node for receiving network messages. The sender process on a node must have a socket connection with each of the receiver processes on other nodes. Thus, the number of socket connections per node on an n-node system is $2 * (n - 1)$.

Figure 3 displays the architecture of the prototype on one of the nodes in the cluster.

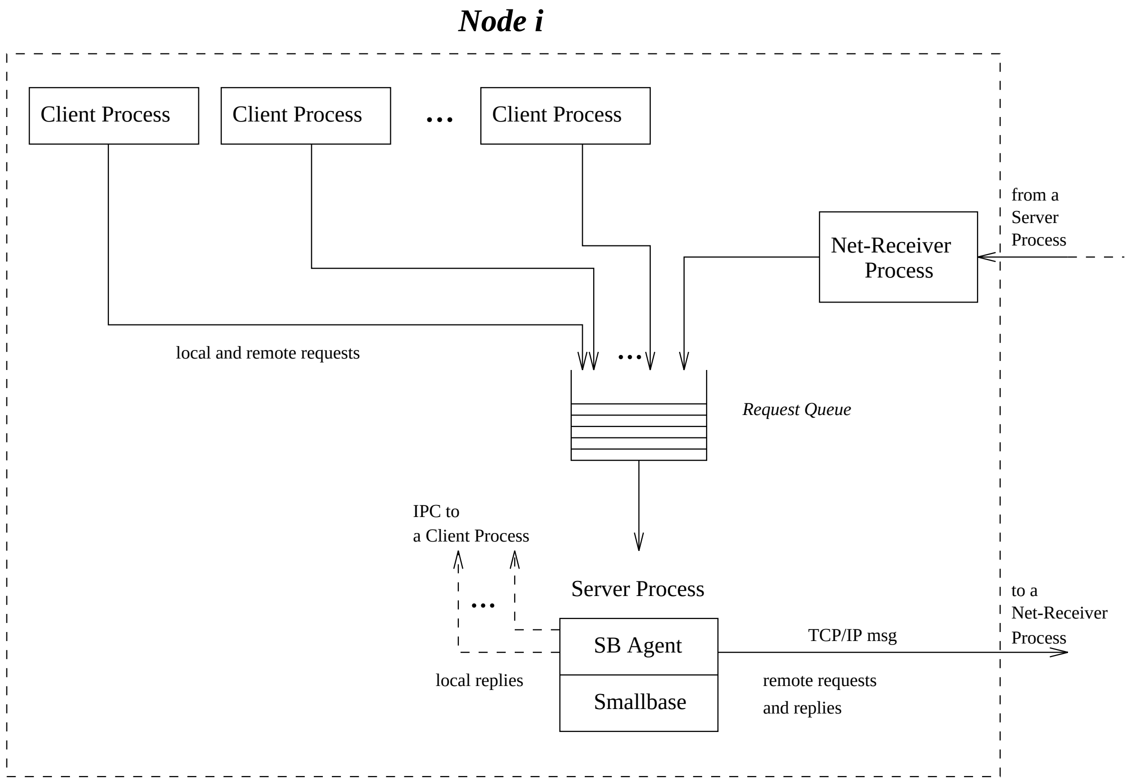

Figure 3: *Architecture*

Running on each node is exactly one Server Process, one Net-Receiver Process, and one or more Client Processes. The Server Process encapsulates the local copy of the Smallbase server and provides transparent support for distributed transactions. It also plays the role of the network sender process. The justification for this decision will be explained below. The Client Processes act as transaction generators. The Net-Receiver Process acts as a support process for distributed transactions. These processes communicate via a queue in shared-memory, namely the Request Queue, and semaphores for local transactions, and via TCP/IP streams for remote transactions. More detailed behavior of each of these components is described below.

Request Queue

All communications to the Server are submitted through the Request Queue. These may be requests from Clients. They may also be requests or replies from remote Servers sent through the Net-Receiver Process on the local node.

The Request Queue is a shared memory segment that is attached to the address space of the Client Processes, the Net-Receiver Process and the Server Process. Client Processes and the Net-Receiver Process enqueue requests/replies, and the Server dequeues them and processes them. Enqueueing a request/reply involves allocating memory for the message, placing the message in the allocated memory and notifying the Server that a new request/reply has arrived. The notification is a V operation on a counting semaphore [18]. Dequeuing a request/reply involves reading a message if one is available. This is a P operation on the counting semaphore. The reader will block if there are no outstanding requests/replies, and will only be scheduled to run by the operating system when a request/reply becomes available.

Client Process

A Client Process is responsible for generating transactions. Since the statements of a TPC-B transaction are well defined, the Client only generates the *AccountID*, *TellerID*, *BranchID*, and *amount* for a given transaction. The Client knows the range of BranchIDs, TellerIDs, and AccountIDs stored at the local node. It randomly draws a *TellerID* and *BranchID* from the ranges stored at the local node and it randomly draws an *AccountID* from the local range with an 85% probability and from the range of a remote node with a 15% probability.

The Client builds a request containing the parameters of the transaction and submits it to the Server Process by inserting it on the Request Queue. The Client Process waits while the transaction is being processed. Upon its completion, the Server notifies the Client. The Client is now ready to submit the next transaction. We chose to make the client requests to the server synchronous to simplify the implementation of the clients and to more closely model a typical interaction of a client with a server.

Since a client does not have more than one outstanding request at a time, we preallocate, for each client, a portion of the Request Queue shared memory segment for submitting requests to the server. Thus, the cost of allocating and deallocating memory on each request is avoided. The client then enqueues the request and notifies the Server by performing a V operation on the counting semaphore associated with the Request Queue. It then waits for the Server to notify it of its completion by performing a P operation on a different binary semaphore that is logically associated with that client. Each node will typically have more than one client on it. The reason for that will be explained below. For a node with m clients, there will be $m + 1$ semaphores, 1 for the Request Queue, and 1 for each of the clients.

Server Process

For each of the SQL statements **(1)** through **(5)**, there is a corresponding *section* already stored in Smallbase. A section is a pre-compiled SQL statement. If the transaction is purely local, statements **(1)** through **(5)** must be executed locally. If the transaction is distributed, statements **(1)** and **(2)** must be executed remotely and statements **(3)** through **(5)** must be executed locally.

A Server Process consists of two modules: Smallbase and a Smallbase Agent (SB Agent). The role of the SB Agent is to allow Smallbase to operate in a client/server model in the presence of local and distributed transactions. For local transactions, it converts individual requests to calls to the proper stored sections in the local Smallbase. For remote transactions, it sends a TCP/IP message to the Net-Receiver Process of the remote node requesting the processing of statements **(1)** and **(2)** of the transaction. It then proceeds to invoke the local Smallbase to execute statements **(3)** through **(5)**. It remembers the state of the transaction so that when it receives a reply from a remote Server informing it of the completion of statements **(1)** and **(2)**, it can declare the transaction completed. The SB Agent will process other requests while the remote portion of a transaction is processed on another node.

The SB Agent is also responsible for notifying local Client Processes and remote Server Processes of the completion of their requests. Local Clients are notified by performing a V operation on their semaphores. Remote Servers are not notified directly, but rather, the SB Agent sends replies through a pre-established TCP/IP connection to the Net-Receiver Process on the node that initiated the request.

Note that the Server Process is not multi-threaded in this prototype. Since transactions do not block for I/O, there was no reason to multi-thread the Server. Rather, the SB Agent reads the next request off of the Request Queue and submits it when the local Smallbase is done servicing the current request. To keep the Server as busy as possible, the SB Agent should not find an empty Request Queue. In running the benchmark, we set up as many Client Processes as is necessary to maximize throughput. There is a tradeoff here as too many client processes can degrade throughput because of operating system overhead, and too empty a queue can degrade throughput by allowing the Server to remain idle. The SB Agent removes all outstanding requests from the Request Queue in one operation in order to dilute the cost of dequeuing, i.e., the cost of invoking the Unix semaphore operation (this is the P operation on the Request Queue counting semaphore). This was shown beneficial in the TPK system [16].

We discuss now the justification for having the Server Process send network messages directly to remote nodes without the intermediary of another process. Initially, we thought we needed an extra process to do the sending on behalf of the Server, so the Server would not have to block while waiting for network send operations. In that scenario, there would have been a separate Net-Sender Process, and the Server would have submitted send requests to it via a send-message request queue similar to the Request Queue. We avoided that scenario by ensuring that the Server never blocks on network send operations. This is preferable to the extra process scenario as the latter requires additional semaphore operations.

To explain how we ensured a non-blocking Server, we first explain some of what happens on a socket send operation. The data sent via a socket send operation is copied by the kernel into a kernel buffer, control returns to the calling process, and the message eventually reaches the destination. With TCP/IP, the message is guaranteed to be delivered. If, when performing the copy operation into the buffer, the kernel finds that the buffer does not have enough free space to hold the message, it will block the sending process while the buffer is being emptied. This, of course, is unacceptable as it severely affects throughput. We solved this problem by making the socket buffer large enough that a socket send would never find a full buffer. We note here that even with small messages, it was easy to fill up a buffer because local transactions are so fast that a socket send could easily bump into the data of prior socket sends that had not yet cleared out of the buffer.

Net-Receiver Process

The Net-Receiver Process is responsible for handling requests and replies from remote Server Processes. Its job is to submit the request or reply to the local Server Process in an efficient manner. This is done through the Request Queue.

The Net-Receiver Process submits requests/replies to the Server in exactly the same way as the Client Processes with the minor difference of memory allocation. While the Clients have a preallocated portion of memory for data passing, the Net-Receiver Process must allocate memory for each request/reply it places on the queue. This memory is later deallocated by the SB Agent. Synchronization for allocation and deallocation of the memory is done via user-implemented latches. These can be implemented in very few instructions [10].

We now explain why we separated the sending of network messages from the receiving of network messages. Recall that TCP/IP supports two-way transmission of data between a pair of processes. Hence we did not really need to have a sender process as well as a receiver process on each node. We could have combined them into a single process for handling all network messages. We saw however how it was advantageous to assign the function of sending network messages to the Server Process because it eliminated the need for the Server to communicate with a sending process. Combining the receiving of network messages with the Server was out of the question because there is no single mechanism the Server could have used for simultaneously waiting on (n-1) incoming sockets and on the semaphore for the Request Queue. Hence, it would have had to poll between checking the semaphore and the (n-1) incoming sockets, and polling would have definitely degraded throughput. Consequently, the two functions for sending and receiving network messages were assigned to two different processes.

Summary

To summarize, we quickly go through the steps of a purely local transaction and through the steps of a distributed transaction. In the case of a local transaction, the client places a transaction request on the Request Queue using a semaphore operation, and then waits to be notified of completion via another semaphore operation. The Server dequeues the request using a semaphore operation. It notes that it is a local transaction. It services the request and notifies the client of completion via the semaphore the client was waiting on. Hence a local transaction requires 4 semaphore operations. Note that one of the semaphore operations is a dequeue operation, which will dequeue all outstanding messages from the Request Queue. While this group dequeue operation does not improve the response time of an individual transaction, it does increase the overall throughput.

In the case of a distributed transaction, the client places a transaction request on the Request Queue using a semaphore operation, and then waits to be notified of completion via another semaphore operation. The Server dequeues the request using a semaphore operation. It notes that it is a distributed transaction. It sends a TCP/IP message to the Net-Receiver Process of a remote node requesting that its remote Server execute statements **(1)** and **(2)** of the transaction. It then executes statements **(3)** through **(5)**, marks the transaction as not completed yet, and proceeds to execute other requests. In the mean time, the request reaches the Net-Receiver Process, which places it on the remote Request Queue via a semaphore operation. The remote Server dequeues the request using a semaphore operation. It services the request and sends a TCP/IP message to the Net-Receiver Process of the originating node to indicate completion of the work. Upon receipt of the message, the local Net-Receiver Process places the message on the Request Queue using a semaphore operation. The Server dequeues the message using a semaphore operation. It notes that the transaction is now complete, and it notifies the client of completion via the semaphore the client had been waiting on. Hence, a distributed transaction requires 7 semaphore operations and 2 TCP/IP operations. Two of the semaphore operations are group dequeue operations.

5 Performance evaluation

5.1 Test environment

We measured the performance of the basic prototype on a cluster of 8 workstations. Each workstation was an HP 9000/735 configured with 144 megabytes of RAM. The 735 is rated at 124 MIPS and 147 SPECmark. The operating system was HP-UX 9.01. The observations we make in this paper are not unique to HP-UX, but are applicable to most implementations of Unix. The workstations were interconnected through both an 10 Mbits/sec Ethernet and an 100 Mbits/sec FDDI network.

The benchmark used was the TPC-B application, with 15% of all transactions being distributed. Each experiment consisted of "warming up" for 10 seconds and cooling down for 10 seconds. The actual test period was 120 seconds. The database was constructed with a single branch for each node in the cluster. There were 10 tellers per branch and 100,000 accounts per branch. The total database size was 16.7 megabytes at each node.

The experiments differed from the benchmark specifications in several ways. First, the ACID properties are not maintained because transaction management is currently missing from Smallbase. Thus, the *commit* statement was dropped from the benchmark and the overhead of logging, checkpointing, concurrency control, and actual commit is unaccounted for. We argued earlier that the lack of transaction management support only affects the actual throughput and response time numbers, but does not affect the scalability of the system. We will pay special attention later on to the response time numbers to ensure that the constraint on response time can be met even in the presence of 2-phase commit.

The second way in which the experiments differed from the benchmark was that we could not scale the database size as required by the benchmark, i.e., we could not have 10MB of memory-resident data for each TPS because our hardware does not support such large physical memories. Finally, instead of appending a history record per transaction, a single history record was updated. The reason is that the history file grows with time and overflows physical memory and causes the data to cease being memory resident.

An important change we made to the benchmark was to modify the constraint that 90% of the transactions finish in less than 2 seconds, to having 90% of the transactions finish in less than 10 milliseconds.

We reiterate now that with a 10 msec constraint on response time, a disk I/O cannot be tolerated in the path of a transaction. Hence, even if Smallbase had support for transaction management, its semantics would have to be different from that of the TPC-B benchmark.

5.2 Expectations

We had some expectations for scalability and response time prior to conducting the experiments. The benchmark specifies that 15% of the transactions are distributed. In an n-node system, each node will execute 85% of its transactions locally and will send 15% of its transactions to the other $(n-1)$ nodes, with the remote transactions uniformly distributed among them. Each node will also execute the remote portion of transactions sent to it from other nodes. Since the 85%-15% break-down is independent of the number of nodes over which the database is distributed, there is no reason why the throughput rate per node should not remain constant, and hence why transactional scaleup could not be achieved. Of course there is a loss in performance in going from a one-node to a two-node system because a one-node system has no distributed transactions at all. However, once the overhead of distribution is paid by a two-node system, this overhead should remain constant, independently of the number of nodes in the system.

As far as response time is concerned, we knew that Smallbase executes a local transaction in roughly 300 μsecs. For a distributed transaction, we saw earlier that 2 TCP/IP messages, and 7 semaphore calls would be required. We ran a few experiments to measure the cost of each of these operations. Sending a TCP/IP message consumes roughly 1 msec from the time it is sent from a process on one node until it is received by a process on another node. This number is largely independent of the network (Ethernet or FDDI) as it is dominated by software and protocol overhead. We also measured the overhead of a semaphore call. It was roughly 55 μsecs if it resulted in another process being scheduled and about 13 μsecs otherwise. Hence we were confident that a distributed transaction could complete in roughly 3 msecs. This was well within the constraint on response time that we had to meet and could easily accommodate the cost of 2 additional TCP/IP messages to account for a 2-phase commit protocol.

It is with these expectations that we started running the experiments.

5.3 Performance of basic configuration

In the basic configuration, a single client on each workstation is submitting requests, and the FDDI network is used for inter-node communication. The experiments were run on several cluster configurations. The number of nodes per configuration varied between 1 and 8 nodes. For each configuration, five experiments were run and the results were averaged. The data collected was the overall throughput of the cluster expressed in number of TPS.

Given the above rough estimates of communication and computation costs, "back-of-the-envelope" calculations lead us to expect about 1400 TPS per node. However, with eight nodes the total throughput was only 600 TPS, with the throughput of each node at 75 TPS. This was well short of the expected 1400 TPS. Worse, the overall average response time was over 13 msecs per transaction, and the average response time for remote transactions was over 85 msec. Clearly this performance was much worse than expected and unacceptable for the SCP.

We instrumented the code in an attempt to account for the time delays for each part of a distributed transaction. It was then obvious that the bulk of the time used by these transactions was spent between the time a TCP/IP message was sent and the time it was received. This suggested an unexpected delay in the actual sending or receiving of the message. After further investigations, we discovered that the TPC/IP protocol attempts to coalesce packets to optimize throughput. To this end, after receiving a send request, it waits for a certain amount of time hoping to receive other send requests. Eventually, it times out and sends the message. The inexplicable delays we were seeing were due to waiting until time out. Fortunately, there is an option that can be set to override this default.

No of nodes	Reg Prio, FDDI		High Prio, FDDI		High Prio, Ethernet	
	Thrpt	Efficiency	Thrpt	Efficiency	Thrpt	Efficiency
1	3272		3163		3163	
2	2265	100%	3784	100%	3596	100%
4	3328	73.5%	7275	96.1%	6743	93.8%
8	5790	63.9%	14254	94.2%	13361	92.9%

Table 1: *Cluster throughput with 1 client per node*

The "TCP_NODELAY" option can be set on a socket and will cause messages to be sent immediately.

We re-ran the experiments with the "TCP_NODELAY" option set on sockets. The first column of Table 1 contains the results of these experiments. With two nodes in the cluster, the average throughput was 2265 TPS. This corresponds to an effective transaction rate of 1132 TPS per node, very close to our rough estimates. We use 2265 TPS as the base expected throughput rate. It is not appropriate to use the 1-node TPS rate as the base rate since the 1-node configuration does not have the overhead of network communication. With 8 nodes, the throughput increased to 5790 TPS. This corresponds to a speedup of 2.56 out of a possible 4. The ratio of speedup over maximum possible speedup is termed *efficiency*. The efficiency of the 8-node configuration was 63.9% out of a possible 100%.

The efficiency results were rather disappointing in that we were expecting close to 100% efficiency. However, the response times now met the specifications — an overall average of 0.53 msec/transaction and about 1.6 msec for each remote transaction. Only 0.03% of the transactions exceeded 10 msecs.

At this point the lack of distributed program analysis tools became a real nuisance. The clocks on a network cluster are not synchronized at a fine enough granularity for a trace to make sense when transactions execute in 300 μsecs. Fortunately, we had developed a visual tool to demonstrate the system. The tool continuously displayed the throughput at each node of the cluster at 100 msec intervals. We promptly noticed that dips in throughput occurred frequently and simultaneously at all nodes of the cluster. Since we were running with only one client per node, this suggested a slow down at one node that caused a convoy and promptly degraded the throughput at all other nodes.

5.4 Performance with enhanced priority

We suspected that the slow down was due to interference with normal Unix background processes. To test this hypothesis, we ran the benchmark with high priority. This was done via the command **rtprio**. For these experiments, the benchmark was run at real time priority 10, which is higher than that of most processes.

The results were much closer to our expectations. They are displayed in the second column of Table 1. With two nodes, the throughput was measured at 3784 TPS and at eight nodes the throughput increased to 14,254 TPS. With a base rate of 3784 TPS, the speedup was 3.77 (out of a possible 4.0) and the efficiency was approximately 94.2%.

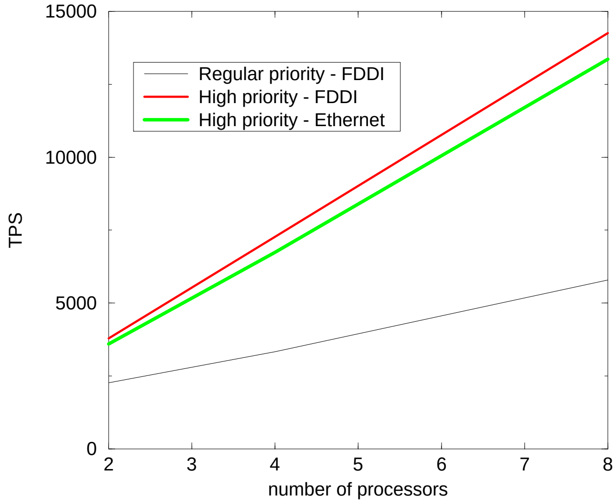

Figure 4: *Cluster throughput with 1 client per node*

The poor performance of the benchmark running at regular priority is attributable to two factors. First, various Unix daemons wake up periodically and perform some tasks. If any such daemon runs for a significant period of time, say 10msec, this would cause both the performance deterioration and the convoys we observed. Running the benchmark at high priority prevents most such daemons from being scheduled. The second, and more important, factor is due to the Unix process scheduling algorithm The benchmark has three classes of processes: servers, clients, and network message handlers. These processes consume CPU cycles in the order in which they are listed above, i.e. the server process executes most of the time, followed by the client processes, and finally by the network message handlers. Under Unix, processes that run for a long time drop in priority. Hence the server process quickly becomes the lowest priority process on a node. It is then preempted by any network message that arrives. This causes a larger number of context switches than necessary, and deteriorates the overall performance of the system.

Although running at high priority provided a big increase in throughput and near-linear scalability, the efficiency of the cluster is still degrading as the number of nodes are increased. In fact, the visual display tool still showed occasional small dips in throughput that were propagated throughout the system. These dips appear more often as the size of the cluster is increased. We intend to do further experimentation to track down the exact cause. We should however emphasize that the efficiency is quite acceptable and should provide adequate scaleup for many SCP configurations.

5.5 Effects of network communication

In this set of experiments, we wanted to test how the speed and the bandwidth of the intercommunication network affect throughput. To this end, we used a 10 megabits/second

No of nodes	1 Client		2 Clients		4 Clients		5 Clients	
	Thrpt	Effic	Thrpt	Effic	Thrpt	Effic	Thrpt	Effic
1	3163		3424		3470		3560	
2	3784	100%	4339	100%	4475	100%	4481	100%
4	7275	96.1%	8388	96.7%	8310	92.8%	8497	94.8%
8	14254	94.2%	15828	91.2%	15560	86.9%	14346	80.0%

Table 2: *Cluster throughput with FDDI network and high priority*

Ethernet instead of the FDDI network. With one client per node and with a real-time priority of 10, the throughput of the system is displayed in the third column of Table 1. As expected, the throughput is lower than with the FDDI interconnect. The difference has to do with the faster speed (lower latency) of the FDDI network and not due to its greater bandwidth. The bandwidth of the Ethernet was adequate for this application. However even with this kind of application, if high-availability is a requirement, log shipping might suffer with Ethernet due to its limited bandwidth.

The results of the three sets of experiments in Table 1 are shown in Figure 4.

5.6 Performance with multiple clients

In these experiments, we were interested in the stability of throughput. As such, we increased the number of clients per node. If resources are underutilized, increasing the number of clients should increase throughput. Once a bottleneck is reached, the throughput should remain stable, while response times rise. If throughput drops precipitously, there is a problem. All experiments were run with a real time priority of ten, and using the FDDI network.

Table 2 and Figure 5 contain the results of these experiments for 1, 2, 4 and 5 clients per node. With two clients per node, the throughput for two nodes was 4339 TPS and at eight nodes it was 15,828 TPS. Thus, throughput increased by about 10% over the case with one client per node. With more than two clients per node, the system becomes bottlenecked by the CPU. As more and more clients are added per node, throughput actually goes down due to extra overhead such as context switching.

5.7 Response time

Table 3 presents detailed response time data for a cluster with 2 clients per node. The overall average response time per transaction is under 1 msec for all configurations. Local transactions take approximately 0.63 msec to complete while remote transactions have a response time of 3 msec or less. Furthermore, fewer than 0.15% of all transactions take more than 10 msec to complete. These response time results leave plenty of room to absorb the overhead of a 2-phase commit protocol and still meet a response time goal of 10 msec.

The same set of experiments were run with the Ethernet network instead of the FDDI network. The results were essentially the same, with an additional 300 μsec for each remote transaction because of the slower network.

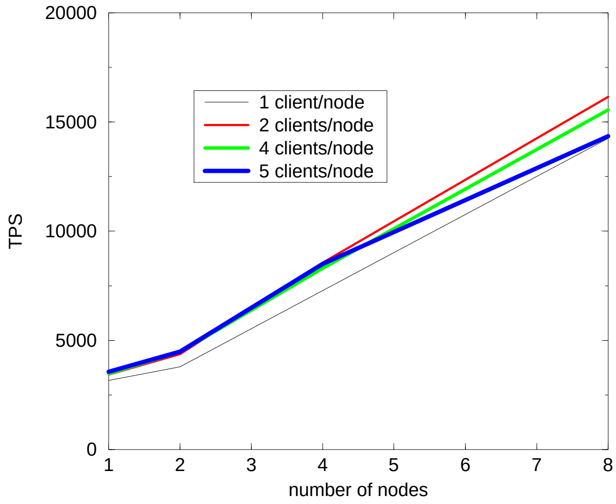

Figure 5: *Performance with multiple clients*

We also gathered performance data for minimum and maximum response times. The minimum response time for a local transaction was about 270 μsec and was about 1 msec for a remote transaction. Somewhat surprisingly, the maximum response times did not vary much between local and remote transactions. In both cases, it was not uncommon to have the slowest transaction take 300–500 msec to complete.

These high response times are consistent with the dips in throughput that we observed in Section 5.4. If for any reason, a delay occurs at one of the nodes, a convoy will quickly form causing the throughput to drop and response times to increase. We have pondered over plausible reasons for these delays and have consulted with our local kernel gurus. At this time, we have no explanation for them. More extensive investigations, such as tracing the kernel may be needed. This is an example of the frustration of tuning such a system without adequate tools for distributed program analysis. Other developers of high-performance, distributed applications on Unix will run into similar frustrations.

In spite of the occasional and up-until-now inexplicable large response times, we should point out that the overall response times are very satisfactory and well within the constraints of the SCP application.

6 Conclusion

We have designed an architecture for a high-performance, distributed OLTP application intended to run on Unix. The application is unique in that it requires the use of a main-memory DBMS and it has severe constraints on response time. We made several decisions in the choice of architecture and IPC. It was mandatory for us to use a reliable message delivery mechanism for distributed communication, hence our choice of TCP/IP. In our

No of nodes	High Priority, FDDI			
	Avg Resp Time (msecs)			% xacts over 10 msec
	Overall	Local	Remote	
1	0.56	0.56	na	0.00%
2	0.90	0.64	2.37	0.05%
4	0.93	0.63	2.64	0.09%
8	0.98	0.63	3.02	0.14%

Table 3: *Average response time with 2 clients per node*

choice of processes and IPC, we were very careful to avoid any polling so that throughput would be optimized.

In our experiments, we learned that it is crucial to understand the scheduling policy of the operating system and to ensure that it does not undermine the performance goal of the application. In our case, because the throughput of the main-memory DBMS is so high, any delay in one node was very quick to propagate to all other nodes of the cluster, causing a convoy and resulting in degradation of the overall throughput. We mostly circumvented this problem by running the application at high priority.

We also learned that system-wide optimizations may interfere with the application requirements. For distributed communication, TCP/IP optimizes for throughput, while our application had response time constraints that did not tolerate the delays introduced in favor of high throughput. Fortunately, it was possible to override the default optimization of TCP/IP.

The lack of distributed programming analysis tools made it very difficult to understand the interplay of a large number of parameters that ultimately contribute to overall throughput. This is crucial if clusters are to become the widely-used alternative to mainframes that many computer manufacturers are advertising.

In spite of the obstacles we encountered, we did obtain very high throughput, low response time and near-linear transaction scaleup on an 8-node cluster running Unix. This was particularly challenging because it was a main-memory DBMS. With a transaction response time of 300 μsec on a single node, any delays are magnified and have a severe adverse effect on scalability.

Acknowledgements

We thank Ravi Krishnamurthy and Henry Cate for their feedback on an earlier draft of this paper. We also thank Scott Marovich, Milon Mackey, and Tim Connors for their troubleshooting expertise. Sherry Listgarten developed the GUI interface for displaying cluster throughput.

References

[1] *Advanced Intelligent Network Release 1, Adjunct Framework Generic Requirements.* Bellcore FA-NWT-001127, Issue 1 October 1991.

[2] A.B. Bondi and V.Y. Jin. Performance Analysis of a Minimally Replicated Distributed Database for Universal Personal Telecommunications Services. *8th ITC Specialist Seminar: UPT*, Santa Marguerita, Italy, Oct 1992.

[3] A. Borr. Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach. *Proc 10th Int Conf on Very Large Data Bases*, Singapore, Aug 1984.

[4] A. Borr. *Personal Communication.*

[5] G. Copeland, R. Krishnamurthy, M. Smith. The Case for Safe RAM. *Proc 15th Int Conf on Very Large Databases*, Amsterdam, 1989.

[6] D.J. DeWitt, J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, vol 35, no 6, June 1992.

[7] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. *Proc ACM SIGMOD Conf*, Boston, MA, June 1984.

[8] M. Eich. Main Memory Database Research Directions. *Proc of the 6th Int. Workshop on Database Machines*, Deauville, France, June 1989.

[9] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[10] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[11] M. Heytens, S. Listgarten, M.-A. Neimat, K. Wilkinson. *Smallbase: A Main-Memory DBMS for High-Performance Applications (Release 3.1).* Database Technology Department, HP Labs, March 31, 1994.

[12] H.-I. Hsiao and D.J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. *Proc 6th Int Conf on Data Engineering*, Los Angeles, CA, Feb 1990.

[13] IEEE Trans on Knowledge and Data Engineering, vol 4, no 6, Dec 1992. Special Section on Main-Memory Databases.

[14] R. King, N. Halim, H. Garcia-Molina, C. Polyzois. Management of a Remote Backup Copy for Disaster Recovery. *ACM TODS*, vol 16, no 2, June 1991.

[15] T.J. Lehman. *Design and Performance Evaluation of a Main Memory Relational Database System.* Ph.D. Dissertation, U of Wisconsin-Madison, Computer Sciences Technical Report #656, Aug 1986.

[16] K. Li and J.F. Naughton. Multiprocessor Main Memory Transaction Processing. *Proc Int Symp on Databases in Parallel and Distributed Systems*, Austin, TX, Dec. 1988.

[17] H. Oliver, J. Carroll, D. Chan, D. Wells. *IN Processor Database Requirements.* Intelligent Network Platform Department, HP Labs, January 17, 1994.

[18] J.L. Peterson and A. Silberschatz. *Operating Systems Concepts.* Addison Wesley, 1985.

[19] R.B. Robrock II. The Intelligent Network–Changing the Face of Telecommunications. *Proc of the IEEE*, vol 79, no 1, January 1991.

[20] W.R. Stevens. *Unix Network Programming.* Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1990.

[21] A.N. Wilschut, J. Flokstra, and P.M.G. Apers. Parallelism in a Main-Memory DBMS: The Performance of PRISMA/DB. *Proc 18th Int Conf on Very Large Data Bases*, Vancouver, Canada, Aug 1992.