The Effects of Parallel Processing on Update Response Time in Distributed Database Design

Jesper Johansson
Boston University

Salvatore March
Vanderbilt University

David Naumann
University of Minnesota

THE EFFECTS OF PARALLEL PROCESSING ON UPDATE RESPONSE TIME IN DISTRIBUTED DATABASE DESIGN

Jesper M. Johansson
Management Information Systems Department
Boston University
U.S.A.

Salvatore T. March
David K. Wilson Professor of Management
Owen Graduate School of Management
Vanderbilt University
U.S.A.

J. David Naumann
Carlson School of Management
University of Minnesota
U.S.A.

Abstract

Network latency and local update are the most significant components of update response time in a distributed database system. Effectively designed distributed database systems can take advantage of parallel processing to minimize this time. We present a design approach to response time minimization for update transactions in a distributed database. Response time is calculated as the sum of local processing and communication, including transmit time, queuing delays, and network latency. We demonstrate that parallelism has significant impacts on the efficiency of data allocation strategies in the design of high transaction-volume distributed databases.

1. INTRODUCTION

Distributed databases (DDBs) have become commonplace as organizations engage in electronic commerce and geographically distributed operations. Design approaches for distributed databases have emphasized retrieval query processing (Apers 1988; Blankinship et al. 1997; Cornell and Yu 1989; Saha and Mukherjee 1994). They often minimize overall system operating cost functions that include local processing and storage and network connectivity. Given the rate at which computing and communication cost are decreasing, response time has become a more appropriate design criterion, particularly for high transaction-volume applications.

Cost minimization approaches to distributed database design typically assign variable costs to CPU time, disk-I/O time, and communication (data transmission) time (Blankinship et al. 1997; March and Rho 1995; Ram and Narasimhan 1994; Rho 1995). The solutions they generate effectively minimize a function that trades off retrieval and update response time due to these factors (plus data storage costs). Since they typically result in overloading less expensive nodes and links (Rho and March 1995), they are appropriate for local area networks (Hevner et al. 1985) or very low-speed wide-area networks. Approaches that minimize response time directly are more appropriate for high-speed wide area networks.

In such networks queuing delays and network latency dominate response time (Johansson 2000). Queuing delays occur when multiple requests must be served by the same server (node). The higher the load on a particular server, the longer a request is likely to wait for service. Hence, when cost minimization approaches overload a less expensive node, response time performance from that node is likely to suffer.

Network latency is the amount of time it takes for a message to move from the sender node to the receiver node. Since messages travel at approximately two-thirds the speed of light once they have been transmitted onto the network, latency is strictly a function of distance (ignoring for the moment the effects of routers and other packet network mechanisms). When transmit speeds are low (e.g., 56Kb/sec) or networks are small (e.g., hundreds of yards) network latency can be safely ignored. However, in high-speed wide-area networks (e.g., 100MB/sec at a distance of 1,000 miles), latency can account for over 90% of the overall network response time (Johansson 2000).

Neither queuing delays nor latency incurs a cost per se. They are pure time components. They do not consume system resources. Hence they are ignored in cost minimization approaches to distributed database design. However, they are crucial to response time minimization approaches. While queuing delays have been included in prior response time models (e.g., Rho and March 1995; Cornell and Yu 1989), latency has not. Furthermore, response time minimization models sum the components of response time even if those components can and should be executed in parallel, i.e., at the same time. For example, local update processing can be performed on all copies of replicated data on different nodes at the same time. Each update incurs local processing time (and costs), but the response time of the update transaction due to local processing is not necessarily the *sum* of the local processing times, as modeled in prior research; rather, by effective utilization of parallel processing, it can be limited to the *maximum* of the local processing times.

Overestimating the update time in this manner makes replication less attractive and can result in the selection of more centralized designs when replicated and distributed designs are significantly more effective, having shorter response times and higher availability.

The remainder of this paper is organized as follows. The next section introduces important network response time concepts. The following two sections discuss parallelism in multi-node update processing and present a model that exploits parallelism to minimize the response time of update transactions in distributed databases with replication. In the last section, we present analytical results demonstrating the effects of parallelism on the selection of efficient distributed database designs.

2. NETWORK RESPONSE TIME FUNDAMENTALS

Response time in a distributed database has two components: local processing time and network response time. Network response time has three components: transmit time, queuing delays, and latency. Transmit time is inversely related to network capacity or bandwidth. It accounts for the time it takes to put data "on the wire." For example, if network capacity is 56 kilobits per second (kbps), and a process needs to send a message of 140 bytes, it would take 140*8/56000 = .02 seconds to transmit the data.

Queuing delays are dependent on the volume of communication from all computers connected to a certain network link. It can be thought of as the time spent waiting for permission to transmit. As network capacities increase, queuing delays decrease. In very high capacity networks, they are negligible.

Latency is the time it takes a signal to propagate from the sending node to the receiving node once it has been transmitted. It is the difference between the time the first bit is transmitted and the time it is received. Latency, in its simplest form, depends on the distance between sender and receiver and on the communications protocol and techniques used. Latency is the inverse of signal speed (SS). The theoretical upper limit on SS is the speed of light. However, in wired networks, SS will always be lower. For example, the propagation speed of an electrical signal in a copper wire is about $2*10^8$ meters per second (Stallings and Van Slyke 1998, p. 119). Protocol and technology considerations may further increase latency.

To highlight the importance of latency, consider a 140-byte message, transmitted for 600 miles. If we assume a signal speed of two-thirds the speed of light, the transmission time in a 56 Kbps network is 20 msec, while latency is 5 msec. By contrast, in a T1 line (operating at 1.544 megabits per second) network, transmission time is only .7 msec, while latency remains 5 msec. In the 56 Kbps network, we would have underestimated network response time by 20% if we disregarded latency. In the T1 example, network response time would have been underestimated by nearly 88%! It is important to realize that the effect of latency is also related to message size. Small messages are more sensitive than large ones, since latency is proportionally a larger component of network response time for small messages.

The coordination required by update operations in a distributed database with replication generates many small messages and increases directly with the number of copies. By contrast, retrieval operations consist of only two messages: a request (typically small), and a response (typically large). In the next section, we present our model of parallelism of update queries in distributed databases with replication. In the following section, we present analytical results demonstrating the advantages of parallelism in multi-node update queries.

3. MULTI-NODE UPDATE PARALLELIZATION

In much of the distributed database (DDB) design research, updates were either ignored (Apers 1988; Blankinship et al. 1997; Cornell and Yu 1989; Saha and Mukherjee 1994) or considered as strictly sequential operations (Lin et al. 1993; Ram and Narasimhan 1994; Rho 1995) as illustrated in Table 1.

Table 1. Gantt Chart of Sequential Lock Granting

Action/time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
Send to node 1											
Send to node 2											
Send to node 3											
Receive ACK from node 1											
Receive ACK from node 2											
Receive ACK from node 3											

An update request is sent to and an acknowledgement received from each node containing the data to be updated—in sequence. The controlling processor must wait for the response from a request before sending the same request to the next node. As illustrated in Table 1, the entire process of sending a request and receiving acknowledgements for data replicated at three nodes is 11 time periods. The node controlling the update is idle for eight of those time-periods waiting for the storage nodes to respond. Such an implementation results in poor resource utilization and poor efficiency. Real distributed databases are unlikely to be implemented in this way. These requests and responses would likely be executed in parallel (Brobst and Vecchione 1999; Hasan 1996; Informix 1999; Oracle 1999). However, parallel operations are difficult to model and have not been considered in prior research. One of the contributions of this research is to explicitly include the effects of parallelism within a global DDB design model, making it more accurately reflect current implementation environments.

When concurrency control is considered, update transactions are considerably more complex. Before an update can be performed, a lock must be granted on all copies to be updated. A two-phase locking protocol (Ram and Narasimhan 1994; Rho 1995) is commonly used:

1. Request a lock from each node involved in the update.
2. Receive locks from all nodes.
3. Perform update at each node.
4. Receive update acknowledgement.
5. Release all locks.

Hence five messages are required per remote copy. The update origination node first sends lock messages. Each storage node answers with an acknowledgement. It then sends update messages. Each is again acknowledged. Finally, it sends release lock messages. These may or may not need to be acknowledged. In addition to the locking mechanism, a distributed database typically makes use of a two-phase commit protocol, to ensure the consistency of the data. This protocol proceeds within the lock protocol as follows:[1]

[1]At this point, for the sake of parsimony, we are ignoring messages incurred as overhead due to the network transmission protocol. This assumption will be relaxed and explained later in the paper.

1. Send "prepare to commit" message to each node involved in update.
2. Receive "prepare to commit" acknowledgements from all nodes.
3. Send "commit message" to each node.
4. Receive "commit" acknowledgement from all nodes.

The two-phase commit would supplant steps three and four of the two-phase lock protocol. Thus, the total number of messages for each node involved in an update is seven: four outbound and three inbound.

Using a sequential approach, the response time (RT) of an update is calculated as the sum of the response times of these steps, as illustrated in Equation 1, where n is the number of nodes at which the data are replicated, $S_{t,i}$ is the send transmission time from the transaction origination node, t, to data storage node i, $R_{i,t}$ is the response transmission time from node i to node t. $S_{t,i}$ includes transmit time and queuing delay. $R_{i,t}$ includes the latency from t to i of the send transmission, all processing at node i, and the complete network response time of the acknowledgement message to node t (transmit, queuing, latency, and receive time).

$$RT = 4 * \sum_{i=1}^{n} S_{t,i} + 3 * \sum_{i=1}^{n} R_{i,t} \qquad \text{Equation 1}$$

4. MODELING PARALLEL UPDATES

Update transactions for replicated data can be parallelized to improve response time. The transaction origination node would need to coordinate outstanding (non-acknowledged) communications with each of the data storage nodes at the same time, e.g., using a mechanism such as threads. If parallelized, the delay for one set of request/acknowledge messages (such as the lock request/acknowledge messages) can be modeled with a max function rather than a sum. For example, a three-node lock request would have the messages shown in Table 2.

Table 3. Messages in a Three-node Lock Process

$\mathbf{S_{t,1}}$	Send lock request to node 1
$\mathbf{S_{t,2}}$	Send lock request to node 2
$\mathbf{S_{t,3}}$	Send lock request to node 3
$\mathbf{R_{1,t}}$	Receive lock acknowledgement from node 1
$\mathbf{R_{2,t}}$	Receive lock acknowledgement from node 2
$\mathbf{R_{3,t}}$	Receive lock acknowledgement from node 3

Table 3. Gantt Chart of Parallelized Lock Granting

Action/time	T1	T2	T3	T4	T5	T6
Send to node 1						
Send to node 2						
Send to node 3						
Receive ACK from node 1						
Receive ACK from node 2						
Receive ACK from node 3						

Using the message structure in Table 2, a sequential update model has a delay of ($S_{t,1}$+$S_{t,2}$+$S_{t,3}$+$R_{1,t}$+$R_{2,t}$+$R_{3,t}$). In a parallel update model, the total delay for the lock process is MAX[($S_{t,1}$+$R_{1,t}$), ($S_{t,1}$+$S_{t,2}$+$R_{2,t}$), ($S_{t,1}$+$S_{t,2}$+$S_{t,3}$+$R_{3,t}$)]. As illustrated in Table 3, the first term ($S_{t,1}$+$R_{1,t}$) is the delay involved in locking the data at the first node. Similarly, the second and third terms, ($S_{t,1}$+$S_{t,2}$+$R_{2,t}$) and ($S_{t,1}$+$S_{t,2}$+$S_{t,3}$+$R_{3,t}$), are the delays incurred from locking the data at the second and third nodes, respectively. This assumes, of course that the update origination node must send lock requests sequentially, but can process receipts in parallel.

In this example parallel processing reduces the time from 11 to 6 time periods (contrasting Tables 1 and 2). It is possible to further reduce response time if the response times of the lock requests/acknowledgements for individual nodes can be predicted. For example, Table 3 shows that node 2 takes longer to respond than the other two nodes. If we know that *a priori*, we can reorder the requests and improve response time as is shown in Table 4. By sending lock requests in decreasing order of response time, those nodes will have more time to respond and the entire operation can be completed in less time.

Table 4. Optimized Parallelized Lock Granting

Action/time	T1	T2	T3	T4	T5
Send to node 2					
Send to node 1					
Send to node 3					
Receive ACK from node 2					
Receive ACK from node 1					
Receive ACK from node 3					

As discussed above, each replica to be updated requires three sets of messages. The first set is the lock request/acknowledgement. The second set is the update request/acknowledgement. The third set is the lock release message. It is reasonable to assume that messages in these exchanges are the same size. The lock request messages are control messages which need to describe the database, tables, username, passwords, etc. that the database engine needs to determine what to lock, and for whom. The same information would need to be included in the update request, and the lock release, for obvious reasons. Furthermore, the acknowledgements would need the same information since the transaction processor handles many simultaneous transactions and must be able to associate messages with transactions.

We assume that all messages in update queries, except for the message that carries the actual data to be updated, have the same size. The message with the update data will carry its payload in addition to the standard message size. Further, we assume that the response times for these messages are invariant within nodes in a multi-node update. In other words, all messages, regardless of content or purpose, from node i to node j, take the same amount of time. However, messages from node i to node k might take a different amount of time. Thus, the total response time of the update can be calculated as shown in Equation 2.

$$Delay = 3 * \left[MAX \Big|_{j=1}^{n} \left[\left(\sum_{i=1}^{j} S_{t,i} \right) + R_{j,t} \right] \right] + \sum_{i=1}^{n} S_{t,i} \qquad \text{Equation 2}$$

In Equation 3, the multi-copy update response time function, n represents the number of replicas being updated. This equation allows us to mathematically model parallelization of multi-copy updates. It does not model the optimization shown in Table 4, however. That optimization must be obtained algorithmically by re-ordering the messages such that they appear in order of descending $R_{j,t}$. The only important factor is that the update takes place at all nodes. The order in which the messages are sent is not important, other than for the purpose of optimization.

The update origination node must perform its send request operations sequentially. Again, send includes queuing delays and transmit time. The receive time was previously defined as the time from the end of a send request operation to receipt of the response. Thus, as discussed above, receive time for each operation includes the following components:

- Latency of the request from the update origination node to the data storage node
- Processing at the storage node
- I/O at the storage node

- Transmission time (transmit time plus queuing delay) at storage node for ACK message
- Latency from storage node to the update origination node
- Receive processing at update origination node

Furthermore, the origination node must have the capacity to receive responses simultaneously. As was stated above, the sending of messages takes place sequentially. However, it is possible, even likely, that the receipt of responses may occur simultaneously. Therefore, it is assumed that all nodes have the ability to cache incoming messages until they can process them. This receive processing time is essentially a demand on the processing capacities of the node and hence is accounted for in the queuing model that describes the processor availability at each node.

5. ANALYTICAL EVALUATION

As Table 4 shows, the optimal parallel solution takes far fewer time-periods than the sequential solution shown in Table 1. However, what kinds of savings are possible in a realistic system? In this section, we will attempt to answer that by calculating the sequential cost of a three-node update and then compare it to the parallelized cost. We will assume a three-node network, interconnected across an IP backbone. For simplicity, we assume that the database servers are connected directly to the backbone, allowing us to disregard the local network delays. This is a realistic approach with large databases that are accessed from many locations. Furthermore, the local network costs are present regardless of whether the transaction is remote or local and, therefore, are uninteresting in the optimization decision.

Consider a network with three database servers, one each in Boston, Massachusetts, Dallas, Texas, and Seattle, Washington, and an update origination node located in New York City. The backbone is operating at between T-1 speed, 1.544 Megabits per second (Mbps), and T-3 speed, 44.736 Mbps. For simplicity, assume that all messages are 140 bytes in length. This includes 20 bytes for the IP header, 20 bytes for the TCP header (Stevens 1994), and 100 bytes for the content of the message. We are also assuming that we are using TCP for transactions (Braden 1992). That protocol allows the payload of the message to be transmitted in the connection setup request, thus eliminating the need for a separate connection setup sequence and resulting in four send messages and three receive message per update, as in the model presented above.

A certain number of CPU cycles are incurred in each message send request. We assume that all nodes have identical processing power at 12 MIPS (Million Instructions Per Second) allocable to this application. In a wide-area network, send operations take between 12,000 and 15,000 instructions (Gray 1988). We use 15,000 instructions in this analysis. Thus, each message send takes 1.25 milliseconds. Receive processing takes the same amount of time. Local query processing is assumed to take 25 milliseconds, including the I/O delays.

The network speeds and transmit times for each node as well as the latencies among all nodes are shown in Table 5. The latencies are actual figures taken from the delays on the AT&T IP backbone on May 5, 2000. Loss on the backbone at that time was 0.0%, hence we assume reliable delivery.

Messages needed for a two-phase locking protocol with a nested two-phase commit are shown in Table 6. QN is the node at which the query originates, SN is the node where the data to be updated is stored. We assume that the query origination node performs all transaction management.

Table 5. Link Parameters

	Network	Transmit	Latency (ms)			
City	Speed (Mb/sec)	Time (ms)	NYC	Boston	Dallas	Seattle
NYC	44.736	0.0250	0	7	37	64
Boston	1.544	0.7254	7	0	44	71
Dallas	20.985	0.0534	37	44	0	66
Seattle	44.736	0.0250	64	71	66	0

Table 6. Distributed Update Messages

Message	Source	Destination	Message
1	QN	SN	Lock Table
2	SN	QN	Lock ACK
3	QN	SN	Prepare to commit
4	SN	QN	Prepare ACK
5	QN	SN	Commit
6	SN	QN	Commit ACK
7	QN	SN	Unlock Table

The messages in Table 6 can be divided into four phases: lock messages, prepare messages, commit messages, and unlock messages (assuming that it is not necessary to wait for acknowledgement of unlock messages). Each of these phases has to be performed sequentially across all nodes involved in the updates. In other words, a message to any node belonging to the lock phase cannot be parallelized with a message from the prepare message. All storage nodes must respond to lock messages before the query can move into the prepare phase. Table 7 shows the time (ms) for send and receive messages for each node.

Table 7. Time (ms) for Send and Receive Messages

City	Message Send	Message Receive
Boston	1.25+0.7254=1.9754	7+1.25+25+1.25+0.7254+ 7+1.25= 43.4754
Dallas	1.25+0.0534=1.3034	37+1.25+25+1.25+0.0534+37+1.25=102.8034
Seattle	1.25+0.0250=1.2750	64+1.25+25+1.25+0.0250+64+1.25=156.7750

Given these calculations, the update should start by sending the message to Seattle, since it takes the longest to respond. The message to Boston should be sent last. Table 8 shows the message round-trip times for each node. (Although intervals in the table are shown as equal length, each represents a different length of time.) The values in each interval show the cumulative time (in milliseconds) from the beginning of the update until the end of the operation specified.

From Table 8 it is a simple matter to calculate the response time of the entire update. From Equation 2 we calculate the parallel response time as 487.70 ms (3 * 158.05 + 4.55) milliseconds. That can be compared with the sequential response time of 927.38 ms (3 * ((1.9754 + 43.4754) + (1.3034 + 102.8034) + (1.2750+156.7750)) + 4.55) milliseconds, which is calculated using Equation 1. Thus parallelizing the three-copy update decreases the response time by over 48%.

Table 8. Message Round-Trip Times

City						
Message send Seattle	1.2750					
Message send Dallas		2.5784				
Message send Boston			4.5538			
Message rec. Boston				48.0292		
Message rec. Dallas					105.382	
Message rec. Seattle						158.05

This can have an extremely significant impact on the performance of a high transaction-volume system where a single update transaction can affect dozens of tables. This approach to update transaction modeling is embedded in a more general approach to distributed database design (Johansson 1999). One of the problems addressed in that work is the determination of an optimal data replication strategy. Replication can significantly decrease retrieval response time, but, as discussed above, can significantly increase update response time. Parallelizing updates can significantly reduce the update response time making it more attractive to utilize replication to gain retrieval performance.

Consider, for example, a retrieval query requiring the data involved in the update transaction discussed above. Suppose its results are regularly required at Boston, Dallas, and Seattle. If the data are replicated, as assumed above, then retrieval response time is minimized. However, this may not be the most effective overall design, depending on the increase in update response time caused by that replication. Table 9 summarizes update and retrieval response times for different replication strategies in sequential and parallel update environments, assuming a query result of 1,000 bytes and 25 ms local processing time for the retrieval query.

As illustrated in Table 9, maintaining a single copy of the data in Boston minimizes the update response time (45.45 ms), but produces significant query response time delays for users in Dallas and Seattle (170.54 ms and 224.54 ms, respectively). Of course, if there is only a single copy of the data, then there is no opportunity for parallel processing in update queries. Depending on the ratio of updates to retrievals, it may be beneficial to replicate the data to reduce retrieval response time. In that case, a model that assumes sequential processing would significantly overestimate the response time of update queries, with results tending toward the selection of a design without replication.

Table 9. Update and Retrieval Response Times (ms)

Replication Strategy	**Update Model**		**Origination Node (100Kb Query)**		
	Sequential	Parallel	Boston	Dallas	Seattle
Single Copy					
Boston	45.45	45.45	0.00	170.54	224.54
Dallas	104.11	104.11	170.54	0.00	173.87
Seattle	158.05	158.05	224.54	173.87	0.00
Two Copies					
Boston & Dallas (Dallas⁺)	451.95	315.60	0.00	0.00	173.87
Boston & Seattle (Boston⁺)	613.75	477.40	0.00	170.54	0.00
Dallas & Seattle (Dallas⁺)	789.05	476.73	170.54	0.00	0.00
Three Copies ⁺⁺					
Boston, Dallas, & Seattle	927.38	478.70	0.00	0.00	0.00

⁺ Location of copy used for remote retrieval query.
⁺⁺ Local copy assumed for retrieval query.

Suppose, for example, that the expected frequency of update is once per second. In the sequential processing model it is virtually infeasible to replicate the data at all three nodes since each update transaction is estimated to take nearly one second to execute (927 ms). Given that the average execution (service) time and the average time between update requests are approximately equal, the average queuing delay would become intolerable. In the parallel update model, such replication is clearly feasible, the average execution time being less than half of the average time between update requests.

Similarly the sequential model virtually eliminates replication in Dallas and Seattle, primarily due to network latency. However, depending on the retrieval frequency at those nodes, such a solution may, in fact, be optimal when parallel update processing is performed.

Given a distributed database design with dozens of locations, hundreds of tables, thousands of update transactions and retrieval queries, the selection of an optimal or nearly optimal data replication design is a daunting task. Utilization of models such as those presented above is crucial to support a database designer in that task.

6. CONCLUSION

A DDB design often includes replicated data because replication speeds query operations. Since replication increases update costs while it decreases retrieval costs, a balanced design depends on tradeoffs between these two types of operations. At current and future network speeds, network latency dominates the response time associated with messages in a distributed database. Update operations require multiple messages to assure database consistency. In contrast, retrievals require only one message exchange per operation. Since updates involve more messages than retrievals, the effect of latency is magnified.

Parallelization of updates can significantly reduce network latency when data are replicated. We have developed a model that demonstrates these benefits. Our model estimates the cost of updates taking place in parallel compared to the cost of the same operations being performed sequentially. These models have been implemented in a general distributed database design approach (Johansson 1999).

To illustrate the role of the model, we have calculated response times for update parallelization using commercially available TCP/IP network parameters in a continental U.S.A. network. In the example, update parallelization achieved a 48% reduction in response time over sequential operations.

This paper extends DDB research in two significant ways. First it accurately represents network latency, a critical component of response time in current high-speed networks that has been ignored in prior DDB research. Second, it models parallelized update transactions on replicated data, demonstrating its response time benefits. We show that parallel processing can be used to reduce update response time, allowing for a more effective use of replication.

Future research needs to empirically analyze the model at hand, as well as integrate it with a model for processing queries in parallel. Furthermore, future research must examine the stochastic properties of query processing. In this paper, it was assumed, for example, that we could predict the roundtrip message time to each node. However, in a realistic situation, this prediction can be more or less accurate. A stochastic model should account for inaccuracies in the prediction based on actual use of network resources. This kind of analysis can also be used to apply stochastic properties to roundtrip times themselves. It is likely that the roundtrip time from one node to another will take varying amounts of time, following some distribution with certain parameters. The roundtrip time is affected by varying factors, such as network congestion, processor loads at source and destination, etc. Furthermore, in this research, it was assumed that each node has a direct connection to each other node with which it communicates. This enables us to ignore the possibilities of router delays between network links. Extending the model to include router delays should be straightforward, but is a necessary component of future work.

References[2]

Apers, P. M. G. "Data Allocation in Distributed Database Systems," *ACM Transactions on Database Systems* (13:3), 1988, pp. 263-304.

Blankinship, R., Hevner, A. R., and Yao, S. B. "An Iterative Method for Distributed Database Optimization," *Data & Knowledge Engineering* (21), 1997, pp. 1-30.

Braden, R. T. "RFC 1379: Extending TCP for Transactions–Concepts," 1992 (http://www.cis.ohio-state.edu/htbin/rfc/rfc1379.html).

Brobst, S. , and Vecchione, B. *DB2 Universal Database Under the Hood*, March 8, 1999 (http://www.software.ibm.com/data/pubs/brobst1.htm)..

Cornell, D. W., and Yu, P. S. "On Optimal Site Assignment for Relations in the Distributed Database Environment," *IEEE Transactions on Software Engineering* (15:8), 1989, pp. 1004-1009.

Gray, J. "The Cost of Messages," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, Toronto, Ontario, Canada, 1988, pp. 1-7.

Hasan, W. *Optimization of SQL Queries for Parallel Machines*, Berlin: Springer-Verlag, 1996.

Informix. *Extended Parallel Option for Informix Dynamic Server*, Informix, Inc., Menlo Park, CA, March 8, 1999.

Johansson, J. M. "On the Impact of Network Latency on Distributed Systems Design," *Information Technology Management*, (1), 2000, pp. 183-194

[2]The following reference list contains URLs for World Wide Web pages. These links existed as of the date of submission but are not guaranteed to be working thereafter. The contents of Web pages may change over time. Where version information is provided in the References, different versions may not contain the information or the conclusions referenced. The author(s) of the Web pages, not ICIS, is (are) responsible for the accuracy of their content. The author(s) of this article, not ICIS, is (are) responsible for the accuracy of the URL and version information.

Johansson, J. M. *Impact of High-Speed Wide Area Network Response Time Dynamics on Distributed Database Design.* Unpublished Ph.D. Dissertation, University of Minnesota, 1999.

Lin, X., Orlowska, M., and Zhang, Y. "On Data Allocation with Minimum Overall Communication Costs in Distributed Database Design," in *Proceedings of the Proceedings ICCI '93: Fifth International Conference on Computing and Information*, Sudbury, Ontario, Canada, May 27-29, 1993, pp. xvi, 587, 539-544.

March, S. T. , and Rho, S. "Allocating Data and Operations to Nodes in a Distributed Database Design," *IEEE Transactions on Knowledge and Data Engineering* (7:2), 1995, pp. 305-317.

Oracle. *Oracle Parallel Server Option for Windows NT*, Oracle, Inc., Redwood Shores, CA, March 8, 1999.

Ram, S., and Narasimhan, S. "Database Allocation in a Distributed Environment: Incorporating a Concurrency Control Mechanism and Queuing Costs," *Management Science* (40:8), 1994, pp. 969-983.

Rho, S. *Distributed Database Design: Allocation of Data and Operations to Nodes in Distributed Database Systems*, Unpublished Ph.D. Thesis, University of Minnesota, 1995.

Rho, S., and March, S. T. "Designing Distributed Database Systems for Efficient Operation," in *Proceedings of the Sixteenth International Conference on Information Systems*, J. I. DeGross, G. Ariav, C. Beath, R. Hoyer, and C. Kemerer, Amsterdam, December 10-13, 1995.

Saha, D., and Mukherjee, A. "An Optimal File Allocation Policy in a Networked Database Management System," *International Journal of Network Management* (4:4), 1994, pp. 218-223.

Stallings, W., and Van Slyke, R. *Business Data Communication*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1998.

Stevens, W. R. *TCP/IP Illustrated*, Reading, MA; Addison-Wesley, 1994.