Computation and Communication in R*: A Distributed Database Manager

BRUCE G. LINDSAY, LAURA M. HAAS, C. MOHAN,
PAUL F. WILMS, and ROBERT A. YOST
IBM San Jose Research Laboratory

This article presents and discusses the computation and communication model used by R*, a prototype distributed database management system. An R* computation consists of a tree of processes connected by virtual circuit communication paths. The process management and communication protocols used by R* enable the system to provide reliable, distributed transactions while maintaining adequate levels of performance. Of particular interest is the use of processes in R* to retain user context from one transaction to another, in order to improve the system performance and recovery characteristics.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; H.2.4 [**Database Management**]: Systems—*query processing, transaction processing*

General Terms: None

Additional Key Words and Phrases: Distributed computation, site autonomy, distributed recovery protocols

1. INTRODUCTION

This article describes the mechanisms used to establish and control a complex distributed computation. The computation is the activity of a distributed database management system. R*, the distributed database management system [21], executes in a (geographically) distributed set of mainframe processors that are connected by a network that supports virtual circuit connections as well as datagram messages. The computation involves multiple processes executing on behalf of a user request and also supports multiple users computing in parallel, each with its own set of processes. In addition, the system supports distributed, recoverable transactions in the presence of process, processor, and communication failures.

First we give a brief description of the function of R* and describe how the distributed computation is organized. Then we explain the computation model implemented by the system and present the communication model assumed by the system. Finally the bulk of this article discusses at length how R* uses processes and communication facilities to implement distributed database access, maintenance, and recovery facilities.

Authors' addresses: IBM San Jose Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1984 ACM 0734-2071/84/0200-0024 $00.75

R* is an experimental system, being developed at the IBM San Jose Research Laboratory, which transparently supports access to data distributed at multiple processing and storage sites. Besides supporting distributed data access, the R* design objective is to provide efficient and reliable data access while maintaining local site autonomy. Data access efficiency is obtained by optimizing data access requests to minimize a weighted sum of local processing, I/O, and communication costs. Optimal data access plans can make use of distributed parallelism by using network flow controls to pace producer/consumer processes at different sites. Data access efficiency is also enhanced by *binding* the optimized query plan to locations and database access paths. The bound query plan can later be executed without analyzing the query semantics, searching for an optimal execution plan, or checking user access authorizations at execution. Site autonomy is supported in R* by requiring each site participating in a query to validate the query, its plan, and the authorization of the user to access the data stored at that site. Autonomy considerations also lead to graceful system growth properties (e.g., it is not necessary to schedule synchronized, off-line activities at multiple sites to add a new database site).

A prime objective of any database management system is to provide recoverable transactions that are *atomic* (i.e., all or nothing) and *durable* (i.e., committed updates survive processor, communication, and storage failures). In the distributed environment, transaction recovery requires specialized protocols to guarantee transaction atomicity because partial failures in which only one site or communication line fails may leave some sites with imperfect knowledge of transaction status at other sites.

2. THE R* COMPUTATION MODEL

The objective of R* is to support efficient execution of user transactions on a distributed database. It is expected that a user session with R* will last for a significant period and comprise the execution of many database transactions. Most of the transactions are expected to be predefined by the application program serving the user, and to be executed repeatedly. For this reason it is important to retain useful state information about the user and the application from one transaction to the next in order to decrease the execution and communication overhead. With this objective in mind we describe how the distributed execution environment is built and controlled.

An R* user initially executes as a single process at the user's home site. The user's application program, which may support an ad hoc query interface or embody a particular set of business transactions, makes calls to the local instance of the R* database management system. If the requested data is stored locally, the local R* accesses the local database. (Note that since an R* process runs on behalf of each user, it must synchronize concurrent database accesses by multiple users.)

When the user's application accesses data stored at another site,[1] the application request is passed to the local R* database manager which determines the location of the data and issues a request to the R* database manager at the

[1] An R* database "site" may occupy the same machine as other "sites," and can be moved to another machine by changing its network name.

appropriate site. All intersite communication in R* is between R* instances at different sites and never between R* and a remote application program. The application interacts only with its local R* instance. This approach makes the distributed database manager responsible for locating and accessing remote data, instead of requiring the user or the application to know where the data is located. An alternative approach used by many distributed computing applications (e.g., TANDEM [20], XDFS [19], and CFS [5]), requires the client (or application) to send messages directly to the correct remote server, which will accept and process the request. By retaining responsibility for communication with remote sites, R* can transparently support database operations which involve data at several sites and can support migration of data from one site to another without requiring changes to the application program or to user queries [12].

When a user requests access to data stored at a remote site, the request must be passed to and executed at the site of the data. Several alternative computation structures are available for implementing the execution of remote data access requests. One approach is to provide a service process which accepts and services requests (messages) from processes at other sites. For example, Tandem device drivers and Pathway Servers [20] queue incoming requests and process them in turn. Since the service process must serve multiple users and since a database transaction many involve multiple data requests, the service process must, essentially, multiplex itself among multiple clients as well as maintain transaction state information from one request to another by the same client. The throughput of such a service process is limited by the delays associated with virtual memory page faults and database I/O. Also, this single-process approach cannot take advantage of multiprocessor architectures.

Instead of handling all remote requests within a single complex process, it is possible to establish a process to service each incoming request. This is the approach taken by the Argus system being developed by Barbara Liskov's group at MIT [13]. XDFS [19], Grapevine [1], and Clearinghouse [18] also allocate a process for each request. Creating and initializing a process to handle each remotely generated request adds overhead to the execution of the request. A more serious drawback is that the sequence of requests within a transaction must somehow be tied together and kept in order. The association of a request with the proper user and transaction will require transmitting the information necessary to make that association with each request. With respect to data access authorization, each request must be individually authenticated. Authentication is a potentially expensive operation.

Instead of providing remote database access service within a single server process or supplying a process per request, R* creates a process for the user on the first request, and retains that process for subsequent use for the duration of the user session. This allows R* to amortize the setup costs over multiple requests that access data at the same site. More importantly, since the process belongs to a single remote user, and since all requests are on behalf of the current user and are part of the current transaction, the R* system does not have to transmit the user identity and transaction identifier with each request. Site to site authentication, user identification, and circuit security can be handled once and for all when the virtual circuit is established. Because the virtual circuit will be used to

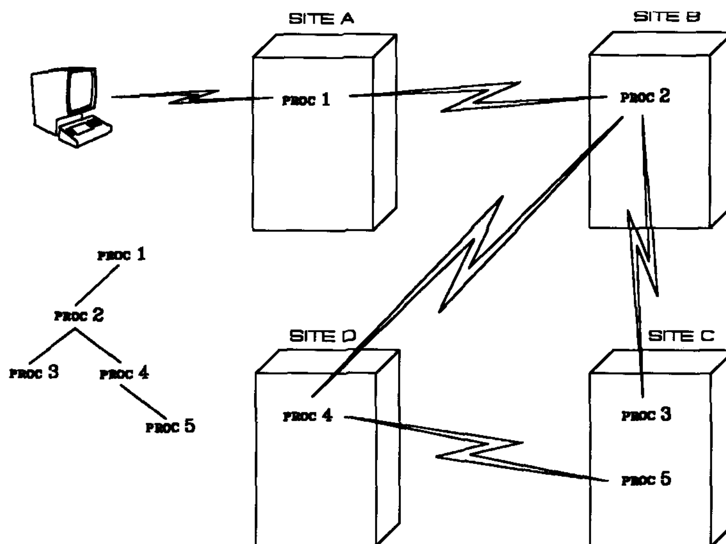

Fig. 1. An R* process tree.

handle all of the computation's communication between two sites, the transaction identifier need only be passed with the first request within each transaction. By retaining processes at remote sites, the user can build up a distributed state which can improve the performance of subsequent access requests. For example, in R* the set of execution plans for the queries and updates embedded in an application program is permanently stored at the sites involved in their execution. When the application program initiates the execution of an embedded query, the query plans for all queries in the application are loaded en masse into the memory of the process. Subsequent requests to execute the same or another query from the application program will proceed without the overhead of loading the query plan.

In R* the requesting process and the process servicing the request are bound together by a virtual circuit. The virtual circuit and server process are created on the first request by an R* process to R* at the server site. The virtual circuit and server process are retained for the duration of the user session, in the expectation that they will be used again. Subsequent access to data resources at the server site, as well as transaction management protocol messages, will use the established virtual circuit and be executed using the existing remote process and its context. Eventually, if the remote service process remains unused, the service process and its resources can be released by closing the virtual circuit connecting it to the requesting process.

In R* some query plans may require that the remote site access data at a third site to complete the original request. In such a case, a request from the remote server process to the third site would establish a new virtual circuit to the third site and a new process serving the remote server would be created. In general, then, a user computation in R* may comprise a *tree* of processes tied together by virtual circuits (see Figure 1). More than one process of the tree may be located

at the same site, due to the request patterns associated with particular queries. Whenever multiple processes belonging to the same process tree exist at a single site, special care must be taken to avoid interference between these processes. The normal transaction-oriented database locking mechanisms are inadequate, since the processes may be participating in the same transaction. When multiple processes from the same process tree exist at a single site, they must share concurrency control and transaction status information maintained by the database access method component of R*. Conflicting use of this shared status data is avoided by prohibiting more than one process of a computation from simultaneously executing in the access method component.[2]

The use of virtual circuits between server and served processes facilitates the detection and resolution of process, processor, and communication failures (see Section 3.1 for a discussion of the virtual circuit failure assumptions). When a process is notified that the virtual circuit to its parent process has failed, it closes the circuits to its subordinates, aborts its part of the current transaction (see Section 4.1 for a discussion of transaction recovery), and terminates. In this way, the subtree of processes below a failure is dismantled. When a process is notified that the virtual circuit to a subordinate has failed, the process initiates abort processing for the current transaction.[3] This strategy for managing the user's process tree retains as much of the distributed computation structure as possible following a failure, while providing a mechanism for releasing processes and resources that are no longer accessible from the user site. The interaction between failure detection and transaction management allows the system to resolve the current transaction so as to retain global database consistency, while continuing to use the surviving local and network facilities for subsequent transactions.

3. THE R* COMMUNICATION FACILITY

This section discusses the communication mechanisms used by R* and the assumptions concerning their failure modes. R* sites communicate using the SNA LU6 protocol [9]. When a process requests activity at another R* site for the first time, a virtual circuit is established with the target site. The first message on that circuit designates R* as the program to be run in the new process established at the target site.[4]

The SNA virtual circuits used by R* are half-duplex circuits in which the direction of message flow is controlled by the processes using the circuit. Processes can send a message of arbitrary length and, optionally, turn the circuit around to receive a response. The virtual circuit insures message ordering and provides flow control to pace the progress of consumer/producer processes when

[2] The Argus group has done some interesting work which addresses this problem.

[3] If the failure is detected during the commit procedure, special actions are taken (see Section 4.1). Otherwise, the abort decision is propagated to the root of the process tree, where it is diffused to the remaining processes which haven taken part in the current transaction. Some subordinate failures can be tolerated without aborting the current transaction.

[4] Currently R* uses the Inter Systems Communication (ISC) facility of CICS [8]. The SNA LU6 protocol as used by ISC passes a CICS "transaction" name designating the program to be run at the target site.

the producer sends multiple messages without waiting for responses from the consumer. After sending a request message and inviting a response, a process can either wait immediately for a response or continue to compute. If it continues to compute, the process can send request messages to other sites (on other virtual circuits) and then wait for each of the responses in turn. In this way, a process can initiate parallel computations at several sites.

Besides the send and receive operations, the SNA virtual circuits used by R* support an out-of-band *signal*, which can be sent by a process at the receiving end of the half-duplex circuit. The signal will be passed (as an exception) to the sending process on its next attempt to send on the circuit. This signal facility is used to escape from "sorcerer's apprentice" mode, when a long response is being sent and the receiver has seen enough.

R*'s principal use of virtual circuits is to implement remote procedure call (i.e., request/response) interactions between sites. A process constructs and sends a message describing the desired function and its parameters, and then awaits the reply. Remote procedure calls are used for most nondata intersite requests. However, when fetching data from remote sites, a bulk data transfer protocol is needed to avoid having to buffer an entire answer set at one or both of the sites involved. Bulk data transfers are explicitly programmed with multiple sends and receives issued to transfer long responses. The data is blocked into buffers and sent when the buffer is full. Received data is processed as it is received, one buffer at a time. Virtual circuit flow control paces the progress of the sending and receiving processes, and overlapped execution is possible as one site prepares message buffers and another site processes received buffers. As mentioned earlier, the out-of-band signal is used to halt data transfer when the remainder of the data is not needed.

R* also uses a datagram protocol for certain functions. The datagram service available in CICS sends an unacknowledged message over a virtual circuit to a remote site which (if the message arrives and the site is up) creates a process running a specified program to handle the message. Datagrams are used by R* to distribute global deadlock detection information and to resolve transactions which experienced failures during the commit process. As discussed in Section 4.1, the failure recovery mechanisms of R* cannot rely upon the virtual circuits used during normal processing, since virtual circuit failures are among the failures handled by the transaction recovery mechanism. Both deadlock detection and transaction recovery are time-triggered, periodic activities. The loss of a message in these contexts is unimportant, since the next cycle of deadlock detection or transaction recovery will resend the messages if they are still needed.

3.1 Virtual Circuit Failures

The management of R* distributed computations and transactions depends critically upon the detection of remote failures. Database recovery mechanisms for local failures are fairly well understood. In general, restart processing following a local failure permits the local database manager to restore the local database to a transaction-consistent state [7]. In a distributed system, however, computations may fail piecemeal. It is important that the nonfailed components of the

distributed computation become aware of the partial failure, and that they react in an orderly and predictable fashion to preserve a globally transaction-consistent database state.

The role of the virtual circuit in detecting and reporting failures of its endpoints or communication path is important to the control of R* computations in the presence of faults. R* cannot use a timeout mechanism for detecting remote failures because it is not possible to bound the expected response time for a given request. Concurrency conflicts, remote site load, and uncertainty about exactly how much processing and I/O are implied by a given request make it difficult to estimate the expected response time. End-to-end failure detection could be implemented by periodically polling the server process status. Such status polling would add considerable complexity to the R* communication service and requires system-dependent coding to determine whether a server process is healthy.

Instead of implementing end-to-end failure detection, R* relies upon the virtual circuit send/receive interface to report an exception if the process or processor at the other end has failed, or if any of the communication lines, modems, or switching nodes has failed. Communication failures are reported to the processes at both ends of the virtual circuit. The virtual circuits provided by the SNA network architecture have these necessary failure detection and reporting capabilities. The virtual circuit protocol implementation (VTAM) uses low-level acknowledgments, line monitoring, and timeout-retransmission to achieve the needed failure detection. The use of fixed routing for established virtual circuits facilitates failure reporting, in that the network can determine which virtual circuits are affected by link and processor failures.

4. R* COMMUNICATION PROTOCOLS

This section discusses the communication protocols used by R* to manage its distributed computations. We first present an overview of the R* system structure. We then discuss the transaction management protocols, particularly those used for recovery from failures prior to or during the transaction-commit procedure. We also discuss R*'s use of communication facilities for distributed authorization and authentication, query compilation and query execution.

The R* system is composed of three principal subsystems: the transaction manager, the communication service, and the database manager (see Figure 2). The transaction manager (TM*) is responsible for orchestrating and controlling distributed transaction commit and transaction recovery from remote failures. The transaction manager is also responsible for local and distributed deadlock detection. The communication service (DC*) provides some added function above the virtual circuit interface. In particular, the communication service keeps track of which virtual circuits have been used during the current transaction. It is also responsible for correctly routing incoming requests to either the transaction manager or the database manager.

The database manager is decomposed into the local database access method and the SQL compiler and runtime component. The local database access method (RSS) provides access to stored relations, concurrency control, and local recovery. The RSS is largely unchanged from the System R base [2] from which it is derived. The SQL compiler and runtime component (RDS) is responsible for

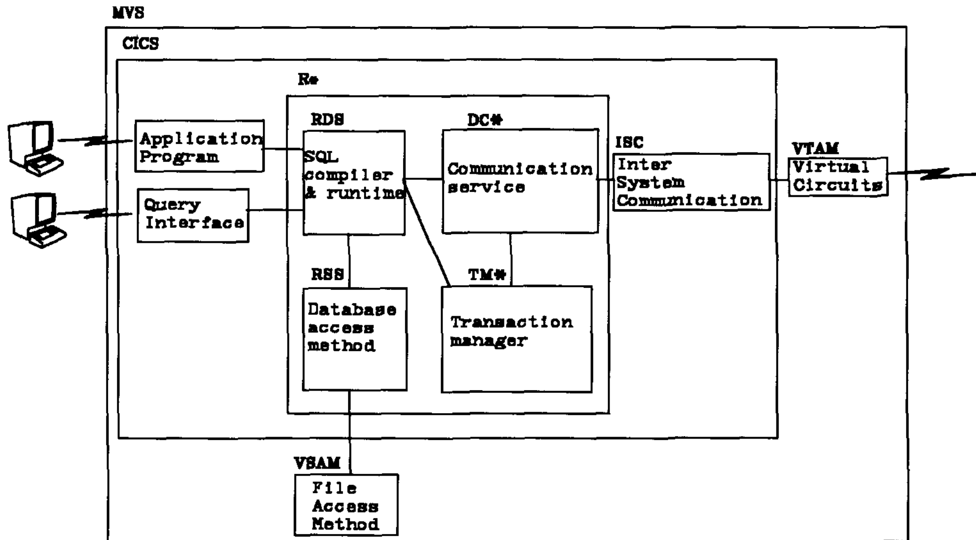

Fig. 2. The R* system structure.

processing statements in the data definition and manipulation language (known as SQL [10]), query planning, and controlling query execution. The RDS generates and processes all messages used to implement SQL language functions.

4.1 Transaction Management Protocols

The transaction manager is responsible for insuring that a distributed transaction is either committed or aborted at *all* sites visited by the transaction. The R* transaction manager uses a nested form of the 2-phase commit protocol [3, 6, 11] to insure that all sites of a transaction come to the same conclusion as to the outcome of the transaction. Transactions are assigned globally unique transaction identifiers by the transaction manager at the root of the process tree. The transaction identifier is communicated to remote processes upon the first use of each remote process within the transaction.[5] The global transaction identifier must be propagated to all sites of the transaction in order to be able to identify the transaction during recovery.

The transaction-commit algorithm used by R* allows any site of the transaction to abort its portion of a transaction unilaterally, *unless* the transaction has entered the commit procedure at that site. In the absence of failures, the commit protocol first propagates a prepare to commit message to all sites of the transaction along the virtual circuits connecting the process tree of the transaction. At each site, the updates of the transaction are made recoverable by writing sufficient information to the nonvolatile (and duplexed) database log to either undo or redo the effects of the transaction. Once a site has become "prepared," it is not allowed to commit or abort the transaction unilaterally, but must await the decision of

[5] The communication service component detects the first use of the virtual circuit within a transaction and informs the local transaction manager. The local transaction manager then sends the transaction identifier to the remote transaction manager.

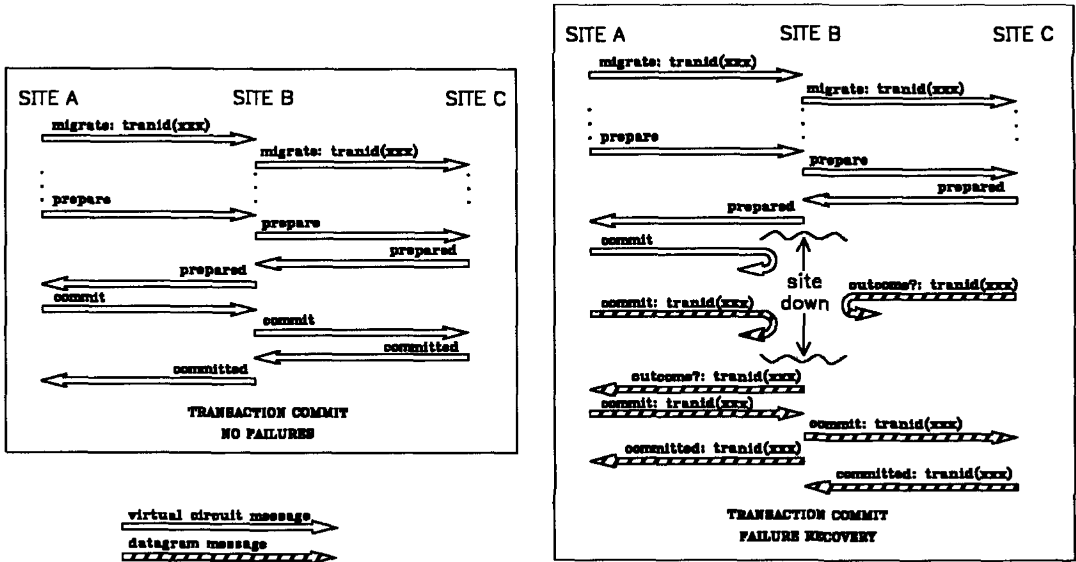

Fig. 3. Commit message flows in R*.

the transaction manager in its parent process.[6] At every level of the process tree the transaction manager propagates the prepare to commit message to subordinate levels and will only acknowledge its own prepare to commit request after receiving positive acknowledgments from all subordinates that participated in the transaction.

When all second-level processes have positively acknowledged the prepare to commit request, the root manager recoverably records that the transaction is committed, informs the local database manager, and sends commit messages to its subordinate processes. Subordinates then pass the commit decision to their local database managers and propagate the decision to lower levels of the process tree.

On the other hand, if all the subordinates do not respond positively to the prepare to commit message, either due to failures (detected by virtual circuit exceptions) or to a unilateral abort reported by some subordinate, the transaction manager will propagate abort messages to its subtree, respond negatively to its parent, and tell the local database manager to abort its part of the transaction. (See Figure 3, which illustrates message flows during commit processing.)

The usual two-phase commit protocol is streamlined in R* to eliminate the second (commit) phase for subtrees which are read-only with respect to data accesses for the current transaction. In this case, a special response to the prepare to commit message tells the parent transaction manager that the subtree is read-only and has assumed that the transaction will commit. Since commit and abort have the same effect at read-only sites, this optimization eliminates

[6] Practical considerations require an external override facility to force the completion of "prepared" transactions when disasters prevent normal recovery.

the messages and delays associated with propagating the commit or abort decision to read-only subtrees, without compromising the reliability of the commit protocol. Since experience suggests that most transactions will be read-only at one or more of the sites visited, this optimization is expected to be quite effective in reducing the commit overhead.[7]

Next, we must discuss the handling of failures during the commit procedure. We have already mentioned (in Section 2) that when a process detects the failure of its parent (or of communication with its parent) and the current transaction is not "prepared" to commit, that process will locally abort the current transaction, sever its connections to subordinates, and terminate. If, on the other hand, the commit procedure is interrupted by process, processor, or communication failures, a recovery procedure is required to resolve the status of outstanding transactions. Since failures may destroy the processes and virtual circuits associated with the transaction, other processes and communication facilities must be used to recover from failures during commit processing.

In R* a "prepared" process that detects the failure of its parent, or a process that is unable to propagate the commit or abort decision to a failed subordinate, will hand over responsiblity for the transaction to the transaction manager *recovery* process at its site. Similarly, the site restart procedures will recover the status of committing transactions and entrust them to the recovery process. The recovery process takes responsibility for resolving the status of "prepared" transactions and propagating commit decisions to subordinate sites. Periodically, the recovery process attempts to resolve the transactions in its charge by sending messages to other sites. Since the status of incomplete transactions is recorded on nonvolatile storage, the recovery process can resume its work after a local failure by analyzing the log records kept by the transaction manager.

The transaction recovery process uses datagrams to resolve the status of outstanding transactions (see Figure 3). The use of datagrams for transaction recovery does not eliminate the need for end-to-end recovery protocols. When a recovery message requires a response, a response datagram message will be sent. Recovery messages include transaction outcome inquiries to the parent site, inquiry reponses (commit or abort), commit/abort propagation to subordinate sites, and commit acknowledgment messages. Upon receipt of an inquiry datagram, the transaction manager will send a commit or abort datagram if the transaction outcome is known. Otherwise, no response is made, and the inquiry will be repeated on the next iteration of the recovery process at the subordinate site. When a site receives a commit (or abort) datagram for a transaction being handled by the recovery process, the commit status of the transaction can be resolved at that site. The local database manager is informed of the transaction outcome, an acknowledgment datagram is sent to the parent site,[8] and, if necessary, commit or abort datagrams are sent to subordinate sites of the transaction.

Besides orchestrating transaction commit, the transaction manager is responsible for detecting local and global deadlocks caused by concurrency controls in

---

[7] A detailed discussion of R* commit protocols can be found in [14].

[8] Acknowledgment is required only for commit messages due to an asymmetry in the R* commit protocol. See [14] for details.

the database manager. Periodically, the deadlock detector solicits information from the local database manager as to which transactions are being delayed by which other transactions. The deadlock detector also asks the communication manager which transactions are waiting for responses or expecting a new request over the virtual circuits that have been used by the transaction. Analysis of this information allows the deadlock detector to detect local deadlocks and to isolate potential global deadlocks. Information describing potential global deadlocks is sent, as datagrams, to deadlock detectors at the appropriate sites. The remote deadlock detector can combine the received potential deadlock information with local information to detect global deadlocks. (See [17] for a complete discussion of the R* distributed deadlock detector.) Because deadlocks are persistent, and because the transaction management component aborts in-progress transactions in response to remote failures, this periodic transmission of potential deadlock information will eventually isolate all deadlocks.

4.2 Database Management Protocols

The upper level of the database manager (the RDS) uses the virtual circuit communication facility in a variety of ways. Communication services are used for intersite authentication and access authorization, query planning and binding, query execution, and the interpretative execution of data definition statements.

R* controls access based upon authorizations to users and groups of users. Authorization to access data is granted to a user at a *specific* site (e.g., user DAVE at SDD is *not* the same authorization entity as user DAVE at PARC). Besides identifying the user who is requesting database access, the system must *authenticate* the identity of the requestor. In R* all intersite requests must be on behalf of a user at the requesting site. Each site authenticates the identify of its local users using passwords. Between sites, the identity of the requesting *site* is authenticated. This prevents any site from exceeding the authority of its local users, despite *any* local efforts to subvert the authorization mechanism.

Intersite authentication and user identification is performed whenever a new virtual circuit is established to another site. In principle, cryptographic techniques could be used for intersite authentication and to establish end-to-end secure (encrypted) communication [16]. Currently, R* performs password-based intersite authentication and communicates over unsecured virtual circuits.

R* query planning and binding is initiated at the root process of the process tree. The query optimizer analyzes the query and computes a plan for executing the query. Query planning considers alternative methods for performing the query and selects a plan which minimizes a weighted sum of estimated processing, I/O, and communication costs. The query analysis makes use of catalog information describing the data to be accessed. Catalog information describing remotely stored data must be fetched from remote sites using a remote procedure call protocol.[9] Once a plan is chosen, it is distributed in parallel to all remote sites involved. At each remote site, the plan is checked and retained locally for

[9] Once the catalog information is fetched, it is cached locally and reused for subsequent query planning. A version numbering scheme allows the detection of obsolete caches during query plan distribution [12].

subsequent execution. Since the plan is sent to all remote sites before collecting their responses, the remote plan processing can be overlapped. At each site, a transaction *save point* is established in order to be able to undo the plan at all sites if some site rejects the plan (e.g., if the plan used obsolete cached catalog information). Transaction save points in R* are a limited form of nested transactions, which allow the local transaction state to be restored when local changes must be undone [7]. If any site rejects the plan, the root process instructs the other sites to roll back to their save points. This use of save points allows query planning to occur as *part* of a transaction that may plan and execute multiple queries.

To execute a query, a request to execute the remote portion of the previously planned query is sent to the remote site. Most query plans specify that the participating sites return a specified (possibly parameterized) record set to the site which requests the execution. After locating the requested plan and performing some authorization checks,[10] the remote site begins to form the answer set and packs the result into answer buffers. Formation of the answer set may involve obtaining answer sets from other sites. As each answer buffer is filled, it is sent on the virtual circuit to the requesting site. At the requesting site, answer buffers are read from the virtual circuit and processed until a buffer indicating that it contains the last records of the answer set is received. This bulk data transfer protocol takes advantage of the virtual circuit flow control and message ordering features to facilitate overlapped processing between the producer and consumer processes.

Occasionally it is necessary to interrupt the generation of answer buffers. This occurs when the consumer process determines that the remainder of the answer set is not needed. In this case, the consumer process sends a signal which is passed, as an exception, to the producer process on its next attempt to send. The communication manager insures that the signal leaves the virtual circuit with the consumer process in send state and the producer ready to receive another request. Figure 4 illustrates the message flows for query planning execution. (See [4] for a detailed presentation of query planning.)

5. DISCUSSION AND CONCLUSIONS

We have presented the distributed computation environment used to support the R* distributed database management system. The tree of processes and the virtual circuits connecting them are used in a variety of ways to support distributed data access for recoverable multisite transactions. The retention of remote processes and the virtual circuits connecting them for the duration of the user "session" improves execution performance whenever repeated accesses are made at a remote site. The use of virtual circuit connections simplifies end-to-end protocol implementations and faciliates failure detection and resolution.

The R* approach to distributed computation can be contrasted with datagram-based and server-oriented distributed systems. Of particular interest in the R* environment is the need to maintain a user and transaction context between requests. The binding of virtual circuits to processes allows the context to be

[10] The authorization checks are performed only on the first request within a transaction.

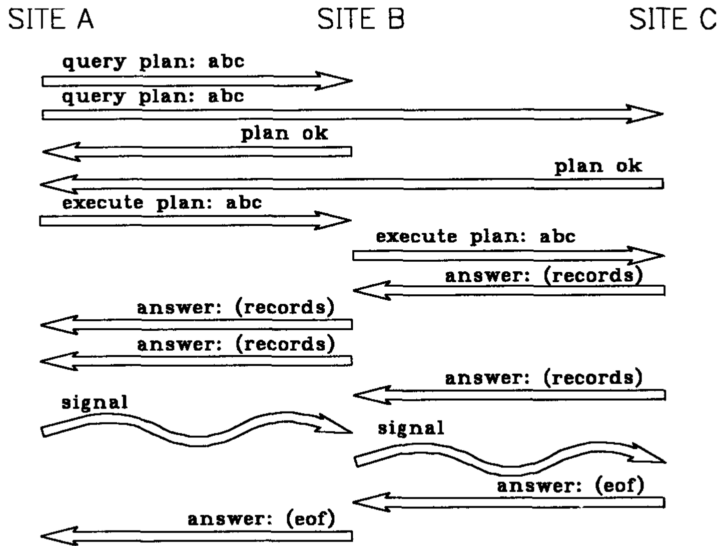

Fig. 4. Query planning and execution message flows.

associated with a process. An alternative approach would be to allow the context to "float" between requests and to identify and bind to the proper context on each request. In some sense, since the virtual circuit implementation must select the correct process for each incoming message, the context selection is simply being implemented at a lower level. The overhead of process creation and context binding must be compared to the virtual circuit management costs to determine the best approach in a given system.

The use of virtual circuits to connect R* processes allows several concerns to be relegated to the network and circuit management implementation. Message ordering and flow control have obvious utility for R*. More fundamental issues are addressed by the failure monitoring implemented by the virtual circuit manager. It is very difficult for R* to place response time bounds on requests issued from one site to another. Database concurrency controls may arbitrarily delay a response, and a request may lead to arbitrarily complex processing at the service site. Therefore, without the process, processor, and communication monitoring associated with the virtual circuit, R* would have to implement timeout-driven inquiry facilities to check on the state of processing of a remote request. This would involve checking the internal state of the service process. Instead, the underlying network manager can detect processor and communication failures in a straightforward fashion, and the operating system can report process failures. Since these features are generally available in modern operating systems and networks, the introduction of higher level failure detection would only increase the complexity of the database management system.

Currently R* is running on multiple processors and is able to perform any SQL query on local or remote data. All data definition statements are operational, as are the global deadlock detection and transaction recovery facilities. All of the protocols described have been implemented and exercised. It is interesting to

note that relatively little effort has been expended on low-level communication issues. Less than a person-year was needed to implement R* facilities above the network interface and below the message generation and interpretation level. This has allowed us to concentrate on the issues and algorithms for distributed query processing and execution.

Early performance measurements (using high-speed communication links) indicate that execution times for some update-intensive simple transactions on remote data are 2 to 4 times slower than their local equivalents. Read-only, single query transactions run between 1.3 and 3 times slower on remote data than on local data. Local transaction execution times are almost the same as execution times in System R [3]. I must emphasize that these performance measurements are preliminary. We expect that system tuning and execution analysis will lead to substantial performance improvements. We are also currently experimenting with R* in operational environments. The results of these experiments will give us a better understanding of the usability and utility of full-function distributed database management systems in business and research.

ACKNOWLEDGMENTS

The implementation of a full-function distributed database management system has not been a simple exercise. The efforts of a dedicated group of researchers were needed to achieve the current level of R* function. C Mohan and Ron Obermarck implemented the distributed deadlock detector. Pat Selinger, Laura Haas, Dean Daniels, Guy Lohman, Pui Ng, Ruth Kistler, and Bruce Lindsay implemented distributed query planning and execution. Paul Wilms implemented the distributed data definition and authorization functions. C. Mohan, Ron Obermarck and Bruce Lindsay implemented distributed transaction management. Bob Yost and Bruce Lindsay implemented process and communication management.

REFERENCES

1. BIRRELL, A. Grapevine: An exercise in distributed computing. *Commun. ACM 15*, 4 (April 1982), 260–273.
2. BORR, A. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proc. Seventh Conference on Very Large Databases* (Cannes, France, Sept. 9–11, 1981), pp. 155–165.
3. BLASGEN, M. W., et al. System R: An Architectural Update. IBM Research Report RJ2581, San Jose, Calif., July 1979.
4. DANIELS, D., SELINGER, P., HAAS, L., LINDSAY, B., MOHAN, C., WALKER, A. AND WILMS, P. An introduction to distributed query compilation in R*. In *Proc. Second International Symposium on Distributed Databases* (Berlin, September 1–3, 1982), pp. 291–309. Also available as IBM Research Report RJ3497, San Jose, Calif., June 1982.
5. DION, J. The Cambridge file server. *ACM Operating Systems Review 14*, 4 (Oct. 1980), pp. 26–35.
6. GRAY, J. Notes on database operating systems. In *Operating Systems-Advanced Course*, Lecture Notes in Computer Science, vol. 60, Springer-Verlag, New York, 1978.
7. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
8. IBM CORPORATION. *CICS/VS System/Application Design Guide.* IBM Form No. SC33-0067-1, June 1978, chap. 13, pp. 279–412.

9. IBM CORPORATION. *Systems Network Architecture—Introduction to Sessions Between Logical Units.* IBM Form No. GC290-1869-1, Oct. 1979.
10. IBM CORPORATION. *SQL/Data System Application Programming.* IBM Form No. SH24-5018, Aug. 1981.
11. LINDSAY, B. G., SELINGER, P. G., et al. Notes on distributed databases. In *Distributed Data Bases*, (Draffan and Poole, Eds.), Cambridge Univ. Press, Cambridge, U.K. (1980), chap. 10, pp. 247–284. Also available as IBM Research Report RJ2517, San Jose, Calif., July 1979.
12. LINDSAY, B. G. Object naming and catalog management for a distributed database manager. In *Proc. Second International Conference on Distributed Computing Systems* (Paris, April 8–10, 1981), pp. 31–40. Also available as IBM Research Report RJ2914, San Jose, Calif., August 1980.
13. LISKOV, B. AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. In *Proc. Ninth ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages* (Albuquerque, N.M., January 25–27, 1982), ACM, New York, pp. 7–19.
14. MOHAN, C. AND LINDSAY B. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proc. Second SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Canada, August 17–19, 1983), ACM, New York, pp. 76–88.
15. MOSS, J. E. Nested transactions: An approach to reliable distributed computing. Ph.D. dissertation, MIT/LCS/TR-260, April 1981.
16. NEEDHAM, R. M. AND SCHROEDER, M. D. Using encryption for authentication in large networks of computers. *Commun. ACM 21*, 12 (Dec. 1978), 993-998.
17. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187–208.
18. OPPEN, D. C. AND YOGEN, K. D. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. Xerox Tech. Rep. OPD-T8103, Oct. 1981.
19. STURGIS, H., MICHELL, J. AND ISRAEL, J. Issues in the design and use of a distributed file system. *ACM Operating Systems Review 14*, 3 (July 1980), 55–69.
20. TANDEM COMPUTERS *Transaction Monitoring Facility (TMF) User's Guide*, Oct. 1981.
21. WILLIAMS, R., DANIELS, D., HAAS, L., LOPIS, G., LINDSAY, B., NG, P., OBERMARCK, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R. R*: An overview of the architecture. In *Improving Usability and Responsiveness* (P. Scheurman, Ed.), Academic Press, New York, pp. 1–27. Also available as IBM Research Report RJ 3325, San Jose, Calif, Dec. 1981.

Received March, 1983; revised August 1983; accepted October 1983