System D: A Distributed System for Availability

S. Andler, I. Ding,* K. Eswaran,* C. Hauser,
W. Kim, J. Mehl, R. Williams

IBM Research Laboratory
San Jose, California 95193

Abstract

System D is a distributed transaction-processing system which addresses the problems of data integrity, system availability, and incremental growth. It is an experimental research vehicle and has been prototyped on a local network of loosely coupled minicomputers. This paper describes the architecture of the system. Novel features of System D include the distributed Resource Manager that provides high availability by diagnosing and taking corrective actions against software-induced crashes, the two-phase Direct Commit protocols that require fewer messages and I/Os than conventional two-phase commit protocols, the datagram-based communications architecture, and the modular storage system.

1. Introduction

System D is a distributed transaction-processing system designed and prototyped at the IBM Research Laboratory in San Jose, Calif., as a vehicle for research into availability and incremental growth of a locally distributed network of computers. The system has been implemented in PASCAL and assembly language on a network of Series/1 minicomputers running the Realtime Programming System (RPS). Six Series/1s are interconnected with a 2Mb/sec insertion ring [HAFN73], which is similar in functions to other local networks, such as token rings [FARB72], slotted rings [WILK79], and the Ethernet [METC76]. The software architecture of System D consists of three functionally distinct modules and instantiations of each of these modules may be replicated in the same or different processors. The modules, whether in the same or different processors, communicate solely via explicit messages.

One important goal of System D was to investigate techniques for achieving high availability in a computing system. Therefore, System D was built with a distributed network of computers which provides a natural opportunity for higher availability than a system running on a single processor. While it is important to preserve the local autonomy of database administrators and users at each computing site of a geographically dispersed system [LIND80b], this notion of site autonomy carries little importance in a locally distributed system. This difference distinguishes System D from geographically distributed systems, such as SDD-1 [ROTH80] and R* [LIND80a, LIND80b, WILL81], and has an important impact on the choice of some of the algorithms used in the system.

Another goal of the project was to develop a software architecture which will facilitate modular growth of a distributed system. The notion of decomposing System D into three distinct modules and replicating and distributing instantiations of any subset of of the modules anywhere in the network is introduced to determine feasibility of modular growth of the software beyond the simple expansion of the network by addition of new processors. The replication and distribution of software modules also enhance availability of the system.

An obvious difficulty with distributed systems is the overhead incurred by the additional communications between processors. Therefore, a simple, relatively efficient message-based communications subsystem was built for interprocessor communications.

2. Overview of System D

The System D software for transaction processing consists of three distinct types of modules: application modules, data manager modules, and storage manager modules. The application module, called A, provides user interfaces for interactive users or application programmers. An application operates on a set of records obtained from a data manager module. An application process specifies the set of records that it is interested in by key association or by positional values. The data manager module, called D, transforms the record-level requests from the A module to page-level requests for the storage module. The record-level requests may be directed to a file of the database or to access paths for the file. The data manager module computes the page numbers in which the desired records reside, and issues a call to the local or remote storage manager module to fetch (or store) pages. All the changes made by an application are kept locally in the data manager module, and are sent to the storage manager by the D module only when the application commits the data changes to stable storage. The storage manager module, called S, supports multiple concurrent transactions against the physical database and database recovery from failures.

We make a distinction between a module and an agent. A module is a function that exists in a node and can be addressed as a logical resource. A module may consist of one or more processes called agents. For example, nodes N1 and N2 may have S modules; i.e. N1 and N2 each have at least one process that can perform the storage manager function. If there are 3 processes (schedulable tasks) in N1 that perform the storage manager function, then there are three S agents in N1.

The System D agents communicate through a message-based communication subsystem (CSS). Any logical resource, whether an agent or a device, can communicate with any other logical resource using datagram messages [BOGG79]. An agent may be addressed by one or more logical resource numbers. A logical resource number is mapped by the CSS into a physical address, consisting of a (processor-id, mailbox-id) pair. Mailboxes may be shared by one or more agents on a processor. Messages can be sent by an agent to any logical resource known to it, and received from any (local) mailbox known to it.

System D guarantees that the effect of running multiple concurrent transactions is equivalent to running transactions according to some serial schedule [GRAY78]. It also guarantees that the serial schedule is the same in all the processors. An interesting variation of the two-phase commit/recovery protocol has been developed for System D. It uses fewer I/Os and messages.

* Current address is ESVEL, Inc., 750A Camden Ave., Campbell, CA 95008.

The notion of a distributed Resource Manager (RM) has been developed and implemented in System D. RM software runs in each processor in the system. Its purpose is to diagnose the causes of failure and to take corrective actions. The RM maintains the system configuration information in a database, called the Resource Manager Database (RMDB). An operator interface to the RM is also provided so that the operator may query and modify the status and configuration of the system: that is, the operator may forcibly shut down system resources or add new resources to the system.

The locations of data and agents are transparent to all the agents in the system (including applications), because of the mapping from logical to physical addresses performed by the CSS. The RM effects reconfiguration of the system by updating the RMDB to reflect changes in logical resource number to physical address associations. When an agent discovers that such an association may be incorrect, it requests the RM in the same processor to verify the association. The RM checks the RMDB and updates the mapping if a change has occurred.

These two features, mapping of a logical resource number to a mailbox and the use of the RMDB to reflect the system configuration, provide location transparency and allow resources to be moved as necessary.

2.1 Transaction Flow in System D

In order to motivate the detailed description of each of the major components of System D in the following sections, a brief discussion of how some of the System D components interact with one another to process a transaction is given here. When a transaction is initiated on node N, the A module binds the transaction to a D module mailbox. Each transaction is completely processed by a single D module.

Disk storage is divided into pages which are numbered; the page numbers span the network of computers. To fetch a data record, each D agent computes the page number needed to obtain the required record; the page number is used to find the S module mailbox to which the page request should be sent.

When a transaction T makes a database request for the first time, it specifies to the D module the maximum amount of time the transaction expects to execute before committing the data at the end of the transaction. When a D agent makes the first page request to an S module on behalf of a transaction, it passes a time value to the S module. The S module which services the first page request is called the master (or coordinator) for this transaction. The master initiates aborting of the transaction if it does not commit within the specified time limit, presuming that the communica-

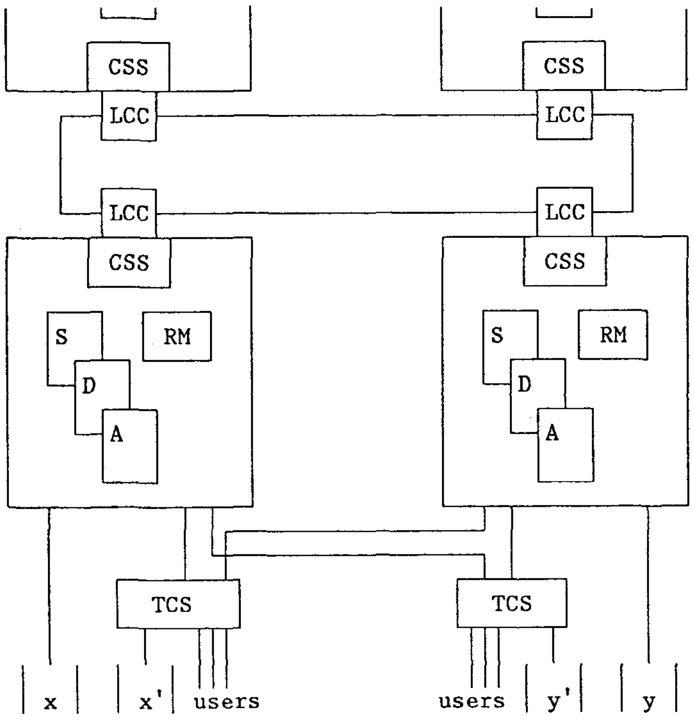

Fig. 1 Typical System D configuration

Fig. 1 shows a typical configuration for System D. Note that the CSS and RM reside in every node. A, D, and/or S modules may appear in any node. The LCCs are the Local Communication Controllers that send and receive messages from the insertion ring. Terminals and shared disks can be connected to a pair of processors through a switch, called a Two-Channel Switch (TCS). The TCS is switched from processor A to processor B under program control, when processor A fails and processor B has to take over the terminals and the disks.

tion medium or the node in which the transaction and/or the D module runs has failed. Whenever the D module makes subsequent page requests for T, it tells the identity of the master to each S module called. The other S modules are slaves with respect to the master for this transaction.

When a transaction is finished, a command is sent to the D module to initiate the commit procedure. All the changes made by T (i.e., the log) have been kept locally in the D module. The D module sends the log to the master S module, which is the commit coordinator for this transaction. The master S module records the information in stable storage and, when this is done, notifies the D module that commit is complete as far as the D module is concerned and that the D module may notify the A module of transaction completion. The master S module (or the D module) then communicates with the slaves to complete commit processing, including actual database changes.

Associated with T, there is also a time limit that T is willing to wait for a lock. If a lock cannot be granted within this time limit, termination of T is initiated. The S module that times out is called the termination initiator for T. The initiator communicates to the D module that there has been a time-out and that the transaction is to be terminated (the presumption is that there has been a deadlock). The D module sends the list of slaves to the master S module and asks that T be aborted. The master S module aborts the transaction and orders all the slaves to abort T. The master S module then communicates to the D module that the transaction has been aborted. The D module does the necessary cleanup and informs the A module that the transaction has been aborted. The transaction may be retried either immediately or after a time delay.

3. Communication Subsystem

3.1 Introduction

The System D communication subsystem (CSS) is a message-based interprocess communication mechanism. The CSS is part of a framework for distributed applications and provides the following main features: (1) simple user-process interface, (2) generic services, (3) location independence, (4) failure detection, and (5) efficient message passing. This section examines these features in detail and describes the architecture of the CSS.

3.2 Main Features

3.2.1 User-Process Interface

The CSS presents a very simple user-process interface, consisting of only a few primitives for sending and receiving messages (Request/Receive/Send), similar to the ones introduced by Brinch Hansen [BRIN70], and a minimum of other primitives for maintaining the mailboxes (Open/Close) and map tables (SetRef/UnSetRef/DeRef) and for initial program load of remote nodes (IPL). The primitives are callable as subroutines from user processes written in PASCAL. There are two routines which may be used to send a message from one logical resource to another: Request and Send. Both routines are synchronous, i.e., they do not return to the caller until the message has been copied out of the caller's address space. One requests a response to the message, while the other simply sends a notice. The Request routine is used to send a request to a remote server and to receive a corresponding answer. A mailbox, in which the answer will be placed, is returned to the caller. The response may be either another request or a notice.

The Send routine sends a message which does not require a response, i.e. a notice or answer, to the specified logical resource. At the receiving end, all arriving messages are queued in storage maintained by the CSS until received by the user process.

The Receive routine allows the user process to receive a message addressed to a particular mailbox. Receiving a message is a synchronous operation. In order to receive a message, the user process must specify the mailbox from which it wishes to receive a message, that is, the mailbox to which the message is to be addressed. If no messages are in the specified mailbox then the caller will wait for the arrival of a message to that mailbox. A "return address" is also returned to the caller which, in the case of a Request, is used as the destination for the answer.

3.2.2 Generic Services

A decoupling between a particular process requesting a service (the "client") and the process(es) performing that service (the "server(s)") is provided by the concept of a "mailbox", to which all messages are sent. A mailbox (also known as a "socket" in other systems [ANDL80]) is unique within a host and must be explicitly opened and closed by a user process. Messages can be received from any specified open mailbox. The CSS also provides "temporary" mailboxes to be used in multi-message communications. This addressing mechanism is completely general. Not only may a single user process have many mailboxes open at the same time, but also many user processes may share a common mailbox.

Fig. 2 shows the use of a shared mailbox to provide a generic service and the use of a temporary mailbox to achieve request/response pairing. The client sends a request message to a globally known mailbox, which is shared by a group of processes, all providing the same service. The message is queued in the mailbox and the client is free to perform other work until it needs the response. Any of the servers can receive the request. After processing the request, the server sends the result to the temporary mailbox that was automatically created for the reply message from the server. The temporary mailbox has a unique number, and will automatically be deleted after being used.

If the server were to return a request (using Request rather than Send), a temporary mailbox will be created that allows the client further communication with the particular server that handled its request. By continuing in this manner, an arbitrarily long connection can be sustained. The following regular expression describes all possibilities:

(Request . Receive)* . Send . Receive

where "." means "followed by" and "*" means "repeated zero or more times". Some important special cases follow:

Send . Receive
Request . Receive . Send . Receive
Request . Receive . (Request . Receive)n . Send . Receive

The first of these examples represents a notice (or hint) that is sent to a process and where it is not crucial that the message is ever received. The second example is the most typical use in System D, and represents a message with acknowledgement, or a remote procedure call. The third example represents a remote procedure call with a transfer of n messages intervening between the initial request and the final response. In either case, the completion of the entire exchange is detected by the final Receive.

3.2.3 Location Independence

A client sends a request to a server using a resource number ("logical" address, or simply "address") for the service it requires. The CSS maps the resource number to a physical address consisting of the processor-id on which the service resides, and a mailbox that has been established (by the RM) for that service. The use of logical addresses allows process location independence. Location independence plays an important role in recovery from failure and dynamic reconfiguration of System D.

The map table that allows the translation from logical to physical addresses is kept in the CSS but is maintained (by calls to the SetRef/UnSetRef/DeRef primitives of the CSS) by the Resource Manager.

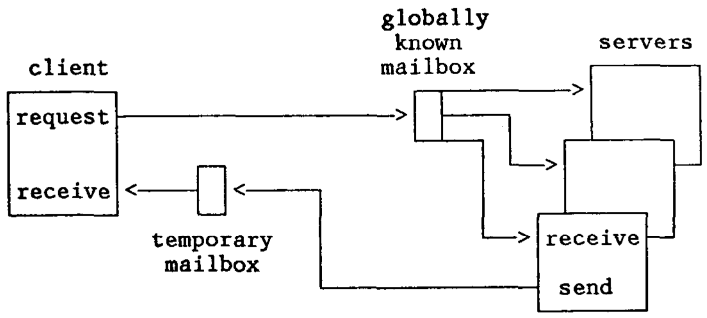

Fig. 2 Shared mailbox and request/response pairing.

3.2.4 Failure Detection

Failure detection in System D is accomplished through time-outs. The CSS provides the mechanism whereby a user process can wait for a message from a mailbox (permanent or temporary), while specifying an upper limit for how long it is willing to wait. If no message arrives in the prescribed time limit, the user process is notified of a time-out. In most cases the user process (e.g., a D or A agent) notifies the Resource Manager of the problem, and the RM attempts to determine the cause of the problem and reconfigure the system as necessary. Depending on the degree of success of the RM, the user process may abort its current transaction or attempt to recover from the failure.

3.2.5 Message Passing

There is often a high overhead associated with message-based communication systems because of the copying of messages from one address space to another and the context switching between tasks. This overhead is especially noticeable when messages are local within a node, i.e., when message passing is compared to more traditional methods of transfer of control and data (i.e. a procedure-call mechanism). The CSS takes care to reduce both the amount of copying that needs to be done and the number of task switches incurred. Copying is reduced by remapping areas of storage from one address space to another, and task switching is reduced by performing the network tasks on behalf of the process that performs a send or receive operation. To allow remapping, message size is limited to the page size of the machine. If a message is short, it is simply copied from the Receive buffer into the receiver's buffer. If a message is long and the user process's buffer is on a page boundary, the page table entries for the Receive buffer and the user process's buffer are swapped.

3.3 Program Structure

The CSS has been designed in a hierarchical fashion, such that each layer provides increased function and hides unnecessary detail at the next lower level.

Level 2 provides the user-process interface (Request/Receive/Send, etc.), and is coded entirely in PASCAL. It implements the abstractions of mailboxes and logical addresses, and performs the mapping from logical to physical addresses. Level 2 also maintains message queues for each mailbox, and supplies the information for the header that is appended to the front of each message.

Level 1, on the other hand, consists of both PASCAL code and routines written in assembly language. It implements the buffering of messages, and distinguishes between long and short messages. The device drivers for the communication device are integrated with the operating system and are written mostly in assembly language.

Level 0 handles the link-level protocol and is implemented by micro-code and hardware in the communication device itself. Level 0 handles such tasks as fragmenting/reassembling of messages that are larger than the frame size on the ring network, hardware acknowledgements of physical delivery of messages, etc. The communication device guarantees that error-free, non-duplicated messages are delivered in sequence to the destination.

3.4 CSS Rationale

In this section we will discuss the rationale for some of the design choices that were made in the CSS, and how they relate to the particular system the CSS was intended to support.

First, mailboxes were chosen as the destination and repository for messages to allow complete de-coupling of client and server processes. A mailbox allows a one-or-many to one-or-many relationship between clients and servers. Send and receive operations on the mailbox are synchronous, but the storage capability of the mailbox allows the processes to be asynchronous.

Second, virtual circuits (or connections), as provided by several proposed or existing networks, were deemed too costly in terms of setup/breakdown effort (negotiating parameters, etc.) for the kind of brief and sporadic communication needs of a distributed transaction system. Typical interactions resemble remote procedure calls to various components of the system, involving a single request message and a single response message. Requests and responses are often small messages, less than a hundred bytes, and sometimes the size of a page of data on secondary storage. Much more seldom a longer transfer is needed, requiring a sustained connection between the two processes involved. Some messages (a.k.a. hints) do not require any confirmation at all, and can be sent as datagrams without requesting acknowledgements. The CSS request-response pairing is especially suited for the implementation of interactions resembling remote procedure calls, but allows simple datagrams in cases where no response is needed and sustained connections at the expense of a little extra bookkeeping by the user when needed for transfer of larger amounts of data.

Third, the CSS does not guarantee delivery of messages but messages, if delivered, will be error-free and appear in the right order. The reasons for not guaranteeing delivery are that every component of a distributed system needs to detect failures of other components, and a mechanism for diagnosing and recovering from failures must exist in the system. Since failures are not very frequent, there is no reason for separately implementing those mechanisms in the communication subsystem (the "end-to-end" argument [SALT81]).

In a distributed system, different hardware and software components have different failure modes. For example, a remote processor may fail while performing a remote procedure call on behalf of a process running on another processor, otherwise unaffected by the failure. It is imperative that the system continues to function in such situations. This requires a mechanism for the caller of a service to detect failures in a remote service (the timeout feature in CSS). When a failure is detected, the system must be diagnosed as to where the error has occurred, so that the faulty path or component can be avoided, the system reconfigured, and the operation re-tried. In System D, this function is handled by the Resource Manager, which is informed of the failure by the user of a service after a timeout.

In a loosely coupled (locally distributed) system of the kind under investigation, communication takes place over local network links with much lower expected failure rates than are typically seen in regular telecommunication networks (e.g., on the order of one error per day or week instead of one per minute). This means that the loss of a message is no more likely than other hardware and software errors, such as a processor failure, memory parity error, or a server program crash, and can be handled by the same mechanism. A separate mechanism to reduce the number of observed failures by guaranteeing delivery of messages is not needed and will cause unnecessary communication overhead. Such a retransmission mechanism, present in most long-haul network protocols, is an optimization for transmission on lines with much higher error rates, involving a large number of store-and-forward hops before reaching the final destination.

4. Resource Manager

4.1 Introduction

The Resource Manager (RM) is a distributed subsystem and resides in each node of the network. Besides preparing a node or the entire system for operation, it is responsible for diagnosing the failures of the System D modules and agents and the operating system on which System D runs, and for taking appropriate actions to recover from the failures. The RM communicates via the communication subsystem (CSS) with System D modules as well as other RMs in performing its functions. The notion of a Resource Manager Database (RMDB) has been introduced in order to provide the RM (and the operator) with system configuration information. The RM consults the RMDB to perform system or node startup as well as to diagnose resource failures. The RM has been designed to provide System D with resiliency to a single failure of a processor or the software (OS, RM, CSS, A, D, or S) running in a processor. The design assumes an installation environment in which each processor is connected to a dual-disk system and in which there is a dual communication path which connects to all the nodes. Under these assumptions, it is sufficient, and efficient, to store the RMDB in only one dual-disk system and have it accessed through one primary-backup processor pair. If the primary processor goes down, the RMDB will be accessed through the backup processor. If one of the disks crashes, the other disk will still have the stable, up-to-date RMDB. The RMDB is a table in which each record contains information about a logical resource (an A, D, or S module): the node-id of the primary processor in which it is to run, the node-id of the backup processor, the status of the resource (UP or DOWN), and the type of the resource (A, D, S, or RM). Other hardware or system-resource data can be maintained in the RMDB in the same way.

4.2 Task Structure

The Resource Manager consists of five tasks: the main (IPL) task, the local message task, the network message monitoring task, the Two-Channel Switch (TCS) task, and the console task. The functions of each of these tasks are described in this section.

4.2.1 Main Task

The main task of the RM is started as part of the Initial Program Load (IPL) of a node of System D. A node may be IPLed manually or remotely. A manual IPL is used to start up one node or the entire system; whereas a remote IPL always starts up only one node, either during system start up or as the last resort by the local message task to recover from software-induced crashes.

The main task reads the RMDB from the node that has access to the stable RMDB and caches it in a local data structure. It then fills in the map table of the CSS (using the SetRef primitive of the CSS) and opens the mailboxes for each of the modules in the node. It then starts each of the agents and all other tasks of the RM.

4.2.2 Local Message Task

The local message task wakes up when it receives a message from an A or D agent that has not received a reply, within a specified time limit, to a request for services to a D or S module. Time-out is the sole mechanism upon which the RM relies to be able to initiate a sequence of actions for failure diagnosis and recovery. Also the local message task is invoked only by an agent residing in the same node: hence the name 'local message' task.

The local message task first locks exclusively the stable RMDB and then reads it. The reason for the exclusive lock is to prevent conflict with the RMDB-read request that the Two-Channel Switch task may make, as will be described below. If the RMDB indicates that the resource in question is now running in its backup processor, the problem must have been detected by another agent already, because the resource has been relocated. This is where the cached RMDB comes in handy. If the cached RMDB entries do not match with the corresponding entries in the newly read stable RMDB, the changes must have been recorded as a result of a local message task running in another node. Then the local message task merely modifies the routing table entry of the CSS, so that all subsequent requests to the resource will be routed to the backup node.

If the cached RMDB entry for the resource in question matches the corresponding entry in the stable RMDB, the local message task attempts to diagnose the cause of the problem by first attempting to establish communication with the node in which the resource is running. The local message task does this by sending an 'ARE-YOU-ALIVE' message to the RM (more precisely, the network message monitoring task of the RM) running in that node. If the response is positive (that is, 'I-AM-ALIVE'), the local message task issues an 'ABORT__TRANSACTION' command to the agent involved. The rationale here is that the service-request message may have experienced a data- or timing-dependent error which may not recur if the transaction under consideration is aborted and resubmitted. Successful processing of the 'ABORT-TRANSACTION' command also serves to indicate that the agent involved probably has not crashed. It is noted here that if the local message task has been invoked as a result of a time-out on a reply to the commit message to the master S module, commit may have taken effect and the transaction may not be aborted. If this is the case, the local message task returns to the wait state.

If the S agent successfully aborts the transaction, the local message task returns to the wait state after notifying the agent which invoked the RM. Otherwise, it will attempt to bring down and restart the agent, since the code or the control structure of the agent may have been destroyed. The request to bring down the agent is routed to the network message monitoring task running in the node where the agent resides. If the agent is successfully brought down by the operating system, probably the code was corrupted. After notifying the invoking agent, the local message task goes back to sleep.

If the agent cannot be brought down, the local message task attempts to shut down and restart the module itself, since the control structures shared by the agents of the module may have crashed. If this is successful, the local message task goes into the wait state.

If the local message task fails to shut down and restart the module, probably the remote RM or the operating system has crashed. Then the node in which the resource in question is running must be remotely IPLed.

Now, if the local message task fails to establish communication with the node in which the resource resides or if the remote IPL was not successful, it will try to communicate with the RM (network message monitoring task) in the backup node of the resource, since the TCS has probably switched. After waiting for a time interval sufficient to guarantee that the TCS will have switched over and the TCS task will have set the TCS-SWITCHED flag (to be explained later), the local message task will query the status of the Two-Channel Switch. The network message monitoring task in the backup node will examine the TCS-SWITCHED flag to determine if the TCS has switched over from the primary node. If the TCS has switched over, the local message task will modify the stable RMDB to reflect the new system configuration and refresh the cached RMDB in the local node. The RM issues a transaction to an S agent to update the RMDB. The RM will also update the map table of the CSS in the local node and, after notifying the agent that invoked it, will go back to the wait state.

If the TCS has not switched over, the local message task will attempt to remotely IPL the node in which the resource resides, if the remote IPL has not already been tried. The reason is that initially it may have failed to communicate with the RM in that node because the RM, the CSS or the operating system in that node may have crashed. If the remote IPL fails, the RM decides that both the primary and the backup processors have failed and notifies the operator to take appropriate measures to prevent any further damage to the database.

4.2.3 Network Message Monitoring Task

The network message monitoring task responds to status queries from the local message task and performs services requested by the local message task. It waits on a message from the local message task or the console task. The local message task may query the status of a remote processor by sending the customary 'ARE-YOU-ALIVE' message to the network message monitoring task of the RM running in that processor. It may query the status of the Two-Channel Switch (that is, whether it has switched to the backup processor) by sending the 'WHERE-IS-TCS' message to the network message monitoring task running in the backup processor. The network message monitoring task also brings down an agent, brings down and restarts a module, and coordinates reintegration of a repaired processor into the system.

4.2.4 Two-Channel Switch Task

The Two-Channel Switch (TCS) task waits on the TCS-event. The Two-Channel Switch has a timer which must be periodically reset by the primary processor to which it is connected. When the primary processor fails to reset the timer, the TCS interrupts the backup processor, which then posts the TCS event. When the TCS task wakes up, it reads the RMDB in order to determine the agents to bring up in the backup processor to replace the agents that have been running in the primary processor. Then it invokes the S-node recovery manager to clean up resources such as lock tables. Finally, it sets the TCS-SWITCHED flag, for use by the network message monitoring task in response to a query from the local message task about the status of the Two-Channel Switch.

4.2.5 Console Task

The console task is provided in order to allow the operator to query and control the system configuration via commands on a system console. The operator may query the system configuration by reading the RMDB. The operator may initiate the integration into (or removal from) the system of a processor (either new or repaired), along with the various agents that will run in it.

4.3 Perspectives

The Resource Manager reflects two important concepts. One is the notion of using a database for keeping system-configuration information to resolve contention among different agents that simultaneously detect the same failure and require the same corrective action, as well as to facilitate orderly reconfiguration of the system resources. The RM reads and updates the configuration database via the transaction capabilities of System D, primarily so that the RM does not have to do the logging and recovery of the configuration database.

Another is the notion of isolating a failure and applying a corrective action suitable for the particular failure that has been diagnosed, rather than simply resorting to the IPLing of a processor. This approach represents a first cut at the seemingly intractable problem of diagnosing and recovering from failures induced by logical errors in software.

The local message task of the Resource Manager is invoked by an A, D or S agent when the agent times out on its request to another agent. Our thesis is that the time-out mechanism detects all failures, be it a deadlock, agent or module crash, communication medium failure or a processor failure. In the current design of the Resource Manager, failures of software or hardware modules are detected only when service requests are directed to them. This passive approach is in direct contrast to the approach adopted in the Tandem NonStop Computing System [KATZ77, KATZ78]. Each processor in the Tandem system must broadcast an 'I-AM-ALIVE' message every 1 second and it also checks for the 'I-AM-ALIVE' message from every processor every 2 seconds [BART78]. If a processor decides that another processor has failed to send it the 'I-AM-ALIVE' message, it initiates recovery actions. The 'active' failure-detection approach of the Tandem system may help to detect a processor failure soon after it occurs even if no service requests are directed to it. It will require minor changes to the Resource Manager to incorporate this feature. However, the problem is just what is meant by 'I-AM-ALIVE'? It is clearly not accurate to say that a processor is 'alive' simply because it can send the 'I-AM-ALIVE' message, when agents running in it may have crashed. Detection of software failures must then rely on message timeouts. It does not appear that the Tandem system attempts to diagnose message timeouts.

Another important difference in the approach to availability taken by System D and Tandem's NonStop Computing System lies in the notion of primary and backup processes. System D supports multiple agents of a module to run in the same processor and the Resource Manager attempts to bring down and restart failed agents while normal service requests may be handled by other agents. In case of a processor failure, agents are brought up in the backup processor and inflight transactions are all aborted. The Tandem approach, on the other hand, is to maintain at all times a primary-backup pair of processes, each in a different processor. The primary process periodically sends checkpoint data to its paired backup process, so that the backup will stand ready to take over as soon as the primary process fails. In the low-end processor environment for which System D has been designed, this difference in design approach does not appear significant. The IPL of a low-end processor normally takes under 1 minute, and this fact must be weighed against the message and processing overhead incurred to maintain the primary-backup pairs of processes. However, in systems that require terminals to be switched and terminal sessions established via a terminal access method, the Tandem approach is desirable.

5. Data Manager

5.1 Introduction

The A and D modules together support simple transaction-processing capabilities. The A module provides a simple interface for interactive users and translates a user transaction into a sequence of record-level requests to the D module. The A and D modules support a transaction which consists of any combination of FETCH, INSERT, DELETE, and ABORT commands sandwiched between BEGIN__TRANSACTION and END__TRANSACTION. In the current implementation, the FETCH, DELETE, and INSERT commands each apply to a single record identified by a unique key value. Also the ABORT command allows the user to abort a transaction so that the effect of the transaction will not be recorded in the database.

5.2 Search and Data Manipulation

The prototype D module only supports record-level requests against single files; that is, no requests which require more than one file to be correlated, such as the relational join, product, or division operations [CODD70], are allowed. Further, the only access path to the stored data that the D module understands is a hashed file, although an interface has also been defined to support B-tree structured secondary indexes to the database.

The D module transforms the record-level request it receives from the A module into a sequence of page-level requests to the S module in order to fetch, insert, or delete a desired record from the database. The address of a record, consisting of the page id of the page in which the record is stored and the byte offset within the page, is obtained by hashing the primary-key value of the record indicated in the user request. In the System D database, records of each file are stored by hashing their primary-key values and those records whose hash values collide are linked on a collision chain. In case of hash collision, both during database loading and during search by the D module, rehash is attempted a few times and then a linear search of the database pages is done to locate an empty slot for a new record (during loading and INSERTing) or to locate a desired record (for FETCH or DELETE). To support the linear search, and record insertion, a bit map is maintained in each data page to indicate empty slot positions.

The D module guarantees serializability of transactions by implementing two-phase locking [ESWA76], which requires that all locks for a transaction be acquired during one phase and that they be released during a second phase without acquiring any new locks. Further, the D module maintains an auxiliary lock table in order to allow maximum concurrency of transactions. The auxiliary lock table contains the page ids of the pages that must be locked until the end of a transaction. When the D module follows the hash-collision chain to locate a desired record, it must first request the S module to lock the page. However, if the desired record is not found in the fetched page, the D module requests the S module to release the lock on the page so that the page may be accessed by other transactions. Before requesting the S module to unlock the page, the D module looks up the auxiliary lock table to make sure the page has not already been locked by the current transaction.

Since System D does not make updates to the data pages until transactions are to be committed, the problem of making updates to the database immediately visible within a transaction must be addressed. For example, if a record has been inserted, the D module must be able to subsequently fetch the record; if a record has been deleted, the D module must return 'RECORD__NOT__FOUND' to the A module in response to a subsequent FETCH command within the same transaction. The D module maintains internal auxiliary data structures, created from the log records, for the inserted and deleted records and the modified bit map entries to give the A module the illusion that updates to the database are immediately and permanently recorded in the database.

5.3 Log Management and Commit/Abort Coordination

Besides responding to each record-level request from the A module, the D module must generate and maintain the log for the transaction for which the record-level request (INSERT and DELETE) is processed. A log record consists of the transaction id, address (page id and byte offset within the page) of the record inserted or modified, the length of the record, and the new value stored. The commit/recovery protocols developed for System D require only the new values to be logged for RE-DOing committed updates. That is, the old values for UN-DOing updates are not necessary. When a record is deleted, the NEXT__POINTER field of the predecessor record is modified to point to the successor record of the record being deleted. When a record is inserted, the NEXT__POINTER field of the current last record on the collision chain corresponding to the key value of the record must also be modified to point to the new record. The bit map entries corresponding to records being deleted or inserted are also updated and logged.

When a transaction is to be committed, that is, END__TRANSACTION is given, the D module sends all the log records for the transaction to the master S module for the transaction. When the master S module forces the log to stable storage and acknowledges this to the D module, the effects of the transaction can be recovered. Then the D module sends to each slave S module to which it made page-level requests only those log records that refer to the part of the database that the S node owns. In order to simplify the task of the S module, the D module sorts the log records in page-id order before sending them to the S modules. As will be shown later on in this paper, it is also possible for the master S module to distribute portions of the log to the slave S modules.

Upon receiving the LOCKWAIT__TIMEOUT message from an S module, the D module sends the master S module for the transaction involved the list of slave S modules to abort the transaction.

6. Storage Manager

6.1 Introduction

The S module receives requests from three sources: (1) from a D module to read a data page from the database, to commit or abort a transaction, etc.; (2) from the Lock Manager or the Commit/Log/Recovery Manager (which are its own subfunctions), for example, to re-process a data page request that has previously attempted to acquire a lock and failed, and (3) from the Resource Manager to fetch or store the RMDB, or to prepare for termination and/or restart of the S module itself or the entire node.

An S module is bound to any D module it services for the duration of a transaction, from the request for the very first data page to the completion of the commit processing, but the binding of an S agent to a D module lasts only while the S agent processes a request it receives from the shared mailbox.

6.2 Program Structure

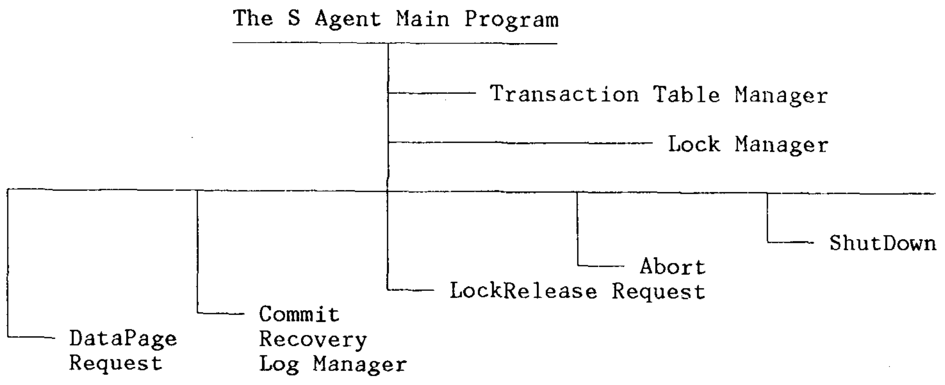

Fig. 3 The S Agent Program Structure

As shown in Fig. 3, an S agent consists of two general service subfunctions, namely a Transaction Table Manager and a Lock Manager, and five specific request handling subfunctions. The general service subfunctions are invoked whenever an S agent is given work to do. The other subfunctions are only invoked to perform specific services, such as to fetch a data page or to manage lock and log data structures. This section describes the operation of each of these subfunctions, except the ShutDown subfunction and the Commit/Log/Recovery Manager. The ShutDown subfunction is invoked to terminate an S module. The Commit/Log/Recovery Manager will be discussed in the context of System D commit/recovery protocols in the next section.

6.2.1 Main Program

Each S agent issues Receive's from the shared mailbox to demand work. It passes the input to specific request handling subfunctions for further processing if it finds work to do; otherwise it goes into a wait state. The subfunctions communicate directly with the requesters through the CSS and return control to the main program only when the requested services have been provided or rejected. An S agent awakens (1) when the shared mailbox is not empty, or (2) when a predefined time limit has elapsed, and it is time to call upon the Lock Manager for deadlock detection.

6.2.2 Transaction Table Manager

As a part of the S module initialization processing, a transaction table is allocated and initialized by the first S agent to be started within the module. Each entry in the table is initially marked as being 'idle' or 'not-in-use'. When the S module is functioning, entries in the transaction table contain information regarding the status and the resources associated with the corresponding transactions. A transaction may be in either active or inactive state after a unique transaction identification number, or transaction-ID, is assigned to it when the very first data page from any S module is requested by a D module. A transaction is active whenever an S agent in the module runs on its behalf.

In a multiple S module environment, it is conceivable that a transaction initiated by a D module requires data pages from more than one S module. Therefore, there is a need to distinguish the master S module and the slave S modules. The master, also known as the commit coordinator, is responsible for (1) creating a unique transaction-ID, (2) writing all log pages to the disk storage during commit processing, (3) assisting with recovery processing of the transaction after a failure, and (4) separating and distributing the updates to the slaves that have access to the partitioned databases.

The slaves, on the other hand, work passively under the direction of the D module and the master S module.

Three linked lists emanate from each transaction table entry: (1) one for the locks held for all data pages referenced by the corresponding transaction from the beginning to the completion of its processing, (2) one for the log pages containing changes associated with this particular transaction, and (3) one for the logical resource numbers of all the slaves that are also bound to this transaction if the S module is the master.

6.2.3 Lock Manager

The System D locking protocol ensures that no concurrent transactions see changes of a transaction until it has safely committed its updates. The Lock Manager guarantees that only requests for locks that are compatible with locks held by other transactions are granted. For example, a shared lock on a page for one transaction is compatible with shared lock on the same page held by other transactions; however, an exclusive lock on a page for one transaction is not compatible with an exclusive lock or shared locks on the same page held by other transaction(s); see [GRAY78] for a discussion of lock compatibility. Requests for incompatible locks are delayed and granted only after all conflicting requests have disappeared. At commit, exclusive locks on changed pages are released only after updates have been applied to the database; this is critical to the commit, logging and recovery techniques adopted in System D. Shared locks are released by the Log Manager subfunction as soon as the D module initiated END__TRANSACTION processing, since no changes need be applied. All locks are released in case a transaction is aborted.

Actions pertaining to a particular transaction are executed sequentially while actions of different transactions can be interleaved on an S module. Whenever the Data-Page Request Handler receives a data-page request from the shared mailbox, it will first request a lock from the Lock Manager. The Lock Manager either grants the lock or timestamps the request and places it on a wait queue. In either case it returns control to the S agent main program, which either proceeds to fetch a page and send it to the waiting D module or processes the next request on the shared mailbox, depending on whether the lock has been obtained or deferred.

Before granting a request the Lock Manager scans its wait queues to see if any transaction has timed out. The Lock Manager also gets periodically prompted by the main program to examine the wait queues. If the S module detects that a transaction has timed out (e.g. because of a global deadlock), it sends a LOCKWAIT__TIMEOUT reply to the waiting D module. In case the S module is the master for the transaction, it also instructs all slaves to abort the transaction.

When a transaction terminates, all participating S modules are asked to release all locks held for the transaction. The Lock Manager selects one or more transactions for execution once it releases the locks they have been waiting on. The Lock Manager re-enters the data-page request to the S module input queue. A processing S agent would see that the lock has been granted and proceeds to fetch the page and send it to the waiting D module.

The reader may note that our use of the time-out mechanism makes it unnecessary to introduce a global deadlock-detection algorithm. In view of the observation that typically less than one transaction in one thousand experiences a deadlock [GRAY81] and because transactions for this type of system would be short transactions, we have decided that it is reasonable to arbitrarily terminate and restart transactions that cannot acquire locks within some time limit. It may perhaps be noted here that the allocation and management of the lock table uses the extendible hashing scheme [FAGI79]. Further, System D supports automatic conversion of shared locks to exclusive locks within a transaction. However, for simplicity, it does not support the intention mode locking described in [GRAY78].

6.2.4 Data-Page Request Handler

The Data-Page Request Handler gets control after a transaction is created or activated by the Transaction Table Manager. It simply reads a page into its storage pool and then sends a copy to the requester if it could successfully acquire an appropriate lock. Otherwise, it suspends processing of the request and leaves it to the Lock Manager to resubmit the request at a later time.

6.2.5 Release-Lock Request Handler

This subfunction is responsible for releasing locks held by a transaction. A lock-release request may be for all locks held by a transaction or only for shared locks. All locks must be released when a transaction is aborted by the master S module or the D module. All shared locks may be released for performance reasons as soon as the D module initiates END__TRANSACTION processing, since shared locks imply no changes to be recorded in the database. Exclusive locks, on the other hand, are released after all the updates of a transaction have been committed. Further, the Release-Lock Request Handler releases locks one at a time, upon request from the D module.

6.2.6 Abort Request Handler

The Abort Request Handler gets control each time there is a specific request to abort. If the transaction is active or it is being processed by another S agent, the Abort Request Handler merely marks the entry for this transaction ABORT__PENDING in the Transaction Table and returns control to the S Agent main program. If the transaction is inactive, it invokes the Lock Manager to release all locks held by the transaction and places the transaction in an idle state.

In addition to a direct invocation from the main program, the Abort Request Handler gets control at two other points. It is called before the Transaction Table Manager activates an entry for a data-page request. It is called immediately after handling of a data-page request is completed and the transaction becomes inactive again. At one of these points, it purges the transaction (i.e. returns it to the idle state), if it is marked ABORT__PENDING.

It is important to note that the implementation of the abort processing is based on the assumption that all S agents are created with equal dispatching priority and that no task switch could occur until a running agent schedules I/O or completes its processing and voluntarily gives up control. Once the Abort Request Handler starts to process an inactive transaction, no other transactions may manipulate the Transaction Table entries until the abort processing is completed. For the same reason, it leaves the responsibility of purging an active transaction to the Data Page Request Handler by branching into the abort processing logic at a later time; therefore, consistency of the transaction status can be adequately maintained.

6.3 Distributed Direct Commit/Recovery

The standard two-phase commit protocol gives each node a high degree of autonomy in deciding the fate of a transaction. In a locally distributed environment like System D, autonomy of the nodes is not so important. Performance is more important. This section describes the variation of the standard two-phase commit protocol developed and implemented in System D. The protocol results in a reduced number of messages and I/O operations. The two-phase locking with exclusive and shared locks is essential for the correct operation of this protocol. Likewise, no "real updates" are performed on the recoverable database objects until a transaction commits. At transaction commit, the transaction's log, maintained by the D module, is sent to the master S module (the commit coordinator).

6.3.1 Direct Commit Protocol (DC)

The six steps of the commit protocol, termed the Direct Commit, follow.

1. When a transaction is ready to commit, the D module sends the log of all the updates performed by the transaction to the master S module, which in turn forces the log onto stable storage.

2. The master then sends "commit" messages to other nodes and begins recording updates to the database it owns. Note that the D module may just as well send "commit" messages to the slave S modules, upon receiving acknowledgement from the master S module that the log has been forced to stable storage.

3. Each slave node then forces its log to stable storage in the order in which transactions committed at that node.

4. Each slave node then acknowledges entry into the committed state to the master with an "ack" message. After this, the node begins making the required database changes.

5. When the master has received all of the acknowledgements from the slaves and has made all of its own updates, it discards its log (returns the transaction to the idle state), and notifies the slave nodes that this has occurred with a "finish" message.

6. Upon receipt of a "finish" message and when done updating its own database, a slave node can dispose of its local transaction log, thus entering the idle state. Note however that, as in other log management schemes, the log for a transaction cannot be disposed of until all previous transactions have been returned to the idle state. Otherwise, at node restart those previous transactions would be redone, erasing the updates of later transactions.

6.3.2 Recovery Protocol

The recovery procedure for this Direct Commit protocol is as follows: Upon restart, a module re-DOes the changes in its local transaction log, and **then and only then** it requests complete logs from any master modules for transactions that are known to them but unknown to the recovering module. Any serial schedule of these transactions is compatible with their earlier concurrent schedule, because the two-phase locking protocol ensures that the updated objects in such transactions must have been locked at the time of node failure. Thus, there can be no interference among such unknown transactions or between them and earlier known transactions. Thus the final recovery operation is to run these transactions serially in any order.

A formal proof of the correctness of the Direct Commit/Recovery protocols will not be given here. Rather, we will illustrate the operation of these protocols by the following example. Consider two transactions, T1 and T2 running on D modules D1 and D2. Modules S1 and S2 store pages X(1..n) and Y(1..n), respectively. T1 touches an X page first so S1 is its master. Likewise T2 has S2 as its master. The two transactions run and commit. That is, each sends its log to its master where the log is recorded on stable storage. At this point, the transactions are committed and must occur uniformly throughout the system: in this case both must occur at S1 and S2. Upon receiving the acknowledgements of commit the D modules distribute the slave log for T1 to S2 and for T2 to S1. Suppose S2 fails in the meantime so that the slave log for T1 is lost.

When S2 is restarted (possibly in a different processor node), it must execute the recovery protocol, bringing its part of the database into a state consistent with the rest of the database. Transactions in the commit process at the time of the failure are completed. Others must be aborted. We first observe that while S2 is not working, S1 is forced to hold the log for T1 since no "ack" message has arrived from S2. Thus, the data needed by S2 to DO T1 is guaranteed to exist in S1. Following the restart protocol, S2 first DOes or re-DOes the changes it finds in its local log -- in this case the changes of T2. An inquiry to S1 then produces the log of T1 for S2. This log is used to DO the changes made by T1 at S2.

At this point, S2 can begin processing new transactions. The database at S2 is in the consistent state it would have had were T2 run before T1. The locking protocol assures us that this is sufficient for consistency since either T2 precedes T1 in the system-wide serial schedule, or the database state at S2 is also compatible with T1 run before T2 (i.e. the two transactions had no conflict in the portion of the database stored at S2). It is of course still possible for unknowing D modules to request actions on behalf of transactions not discovered during this recovery processing. Such transactions will be told to abort either by the S2 or by their own master. They are never allowed to commit, since their locks at S2 were lost and consistency can no longer be guaranteed.

6.3.3 Perspectives

We have also developed a variant of the DC. This protocol replaces the synchronized log disposal of DC with a begin-transaction record in the log of each participant. Before doing anything on behalf of a transaction a node recoverably records the begin record. Commit proceeds exactly as before through step 4. At step 5, the master now discards its log without sending any message to the slaves. At step 6, a slave may discard its log of a transaction as soon as it is locally done with the updates.

The recovery protocol replaces the request to all the masters for "any transactions that are unknown" with specific requests to the masters for "update records for transactions which are known to have begun, but for which no commit record has been written". Since the the local log provides no order information about these updates, they are done after the changes specified in the local log, and in any order among themselves, just as in the DC recovery protocol. The locking protocol insures non-interference.

If n denotes the number of nodes participating in a transaction, the DC protocol has 2n recoverable state changes per transaction (i.e. 2 per node) and requires 3(n-1) messages, n-1 each of "commit", "ack", and "finish". Of these, only the n-1 "commits" are time critical in the sense that database resources are tied up until they are delivered. The "ack" and "finish" messages are used only to dispose of logs (recover log space), so they can be batched to reduce network traffic if necessary.

The variant protocol replaces one set of messages (the "finish"es) with a recoverable state transition at transaction begin. Again, only the "commit" messages are time critical.

Contrast these numbers with two-phase commit where about 3n messages are needed just to inform all the nodes that a transaction is to be committed. Thus, each message is time critical, since resources are tied up until they have all been delivered. The fourth set of messages in two-phase commit is used for return to idle state and is not time critical.

Concluding Remarks

In this paper we have described the architecture of System D. The system was intended to be a vehicle for research into availability and modular growth of a transaction-processing system. We believe that the notion of decomposing a transaction-processing system into the application, data manager and storage manager modules and distributing any subset of these modules to different processors is an interesting contribution of System D. Running multiple agents of any of these modules in the same processor, where each of the agents resides in a different address space, provides an alternative to the current approach to crash recovery which is based on maintaining a primary-backup pair of processes.

The distributed Resource Manager has considerably enhanced our understanding of system failures and availability. The Resource Manager ideas provide a first-cut solution to the baffling problem of detecting crashes induced by software failures. We have also introduced the notion of using a database to maintain system-configuration information to drive the failure-recovery actions of the Resource Manager and to facilitate system reconfiguration.

Further, to support multiple concurrent transactions, a variation of the standard two-phase commit protocol and its accompanying recovery protocol were developed. This new set of protocols appears to be highly effective for the locally distributed network environment like System D, where, unlike a geographically dispersed network, autonomy of the processors is not important.

We designed and implemented a communication subsystem which supports, with a small number of primitives, all the requirements of a distributed transaction processing system designed to provide high availability. In particular, the communication subsystem supports generic services through the use of shared mailboxes, location independence through mapping of logical resource numbers to corresponding physical addresses, and failure detection through user-supplied time-outs.

What we have learned from our work on System D point to a number of interesting problems that warrant further research. As with other systems designed for high availability, System D was designed to provide resiliency to single failure of software or hardware modules. An investigation of the robustness of the conventional single-failure assumption is certainly of value. It is also important to more systematically address the problems of detecting failures and recovering from failures caused by logical errors in software. The knowledge gained from such studies may serve as a basis for establishing methodology for developing more robust software in the first place.

The CSS is being expanded for use in an office system internetwork. Future work includes transferring the CSS to other machine architectures (e.g., the IBM System/370), and extending it for internetworking by providing gateways and other provisions for naming, addressing, and routing functions.

More work could be done on decomposing software into functionally distinct modules. Although we initially liked the idea, we did not examine in depth the tradeoffs and limitations of such an approach.

Acknowledgements

Doug Terry, as a Graduate Student Associate from the University of California at Berkeley, participated in the design and implementation of the Communication Subsystem. While visiting the IBM San Jose Research Laboratory as a post doctoral scientist, Tom Neumann of the Technical University at Darmstadt, W. Germany implemented the loader and the initial version of the Lock Manager. Other contributors to the System D project include Felix Closs, Dean Daniels, Cesare Galtieri, Ralph Pipitone, and William Osborne.

References

[ANDL80] Andler, S., D. Daniels, and A. Spector. On Enhancing Local Network Communication Devices. in Proc. IFIP WG 6.4 Intl. Workshop on Local-Area Computer Networks, Zurich, Switzerland, Aug. 1980. (also available as IBM Research Report: RJ3094, March 1981)

[BART78] Bartlett, J.F. A NonStop Operating System, Proc. 1978 Hawaii Intl. Conf. on System Sciences, Jan. 1978.

[BOGG79] Boggs, D.R., J.F. Shoch, E.A. Taft, and R.M. Metcalfe. PUP: An Internetwork Architecture, Xerox PARC Technical Report: SSL-79-10, July 1979.

[BRIN70] Brinch Hansen, P. The Nucleus of a Multiprogramming System, Comm. ACM, vol. 13, no. 4 (April 1970), pp. 238-241, 250.

[CODD70] Codd, E.F. A Relational Model of Data for Large Shared Data Banks, Comm. ACM, vol. 13, no. 6 (June 1970), pp. 377-387.

[ESWA76] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger. On the Notions of Consistency and Predicate Locks in a Relational Database System, Comm. ACM, vol. 19, no. 11 (Nov. 1976).

[FAGI79] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H.R. Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Transactions on Database Systems, Vol. 4, No. 3, Sept. 1979, pp. 315-344.

[FARB72] Farber, D.J., and K.C. Larson. The System Architecture of the Distributed Computer System--The Communications System, Proc. Symposium on Computer Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972, pp. 21-27.

[GRAY78] Gray, J. Notes on Data Base Operating Systems, IBM Research Report: RJ2188, Feb. 1978.

[GRAY81] Gray, J., et al. A Straw Man Analysis of Probability of Waiting and Deadlock, IBM Research Report: RJ3066, Feb. 1981.

[HAFN73] Hafner, E.R., Z. Nenadal, and M. Tschanz. A Digital Loop Communication System, ICC '73, June 1973. Revised version in IEEE Trans. on Comm., June 1974, pp. 877-881.

[KATZ77] Katzman, J.A. System Architecture for NonStop Computing, Proc. CompCon, Feb. 1977, pp. 77-80.

[KATZ78] Katzman, J.A. A Fault Tolerant Computer System, Proc. 1978 Hawaii Intl. Conf. on System Sciences, Jan. 1978.

[LIND80a] Lindsay, B. Object Naming and Catalog Management for a Distributed Database Manager, IBM Research Report: RJ2914, August 1980.

[LIND80b] Lindsay, B., and P.G. Selinger. Site Autonomy Issues in R*: A Distributed Database Management System, IBM Research Report: RJ2927, September 1980.

[METC76] Metcalfe, R.M. and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks, Comm. ACM, vol. 19, no. 7 (July 1976), pp. 395-404.

[ROTH80] Rothnie, J.B., Jr., et al. Introduction to System for Distributed Databases (SDD-1), ACM Trans. on Database Systems, vol. 5, no. 1 (March 1980)

[SALT81] Saltzer, J.H., D.P. Reed, and D.D. Clark: End-to-End arguments in system design. Second International Conference on Distributed Systems, Versailles, April 1981.

[WILK79] Wilkes, M.V., and D.J. Wheeler. The Cambridge Digital Computer Ring, Proc. Local Area Communication Network Symposium, NBS, May 1979.

[WILL81] Williams, R., et al. R*: An Overview of the Architecture, IBM Research Report: RJ3325, Dec. 1981.