DRA / LANGLEY

NASA Grant NSG 1471

The Embedded Operating System Project

Mid-Year Report, May 1984

Principal Investigator
Roy H. Campbell

Research Assistants
Jeff Donnelly
Raymond B. Essick
Judith Grass
Dirk Grunwald
Pankaj Jalote
David A. McNabb

Software Systems Research Group
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801-2987
(217) 333-0215

NASA Grant NSG 1471

The Embedded Operating System Project

Mid-Year Report, May 1984

Principal Investigator
Roy H. Campbell

Research Assistants
Jeff Donnelly
Raymond B. Essick
Judith Grass
Dirk Grunwald
Pankaj Jalote
David A. McNabb

Software Systems Research Group
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801-2987
(217) 333-0215

ABSTRACT

This progress report describes research towards the design and construction of embedded operating systems for real-time advanced aerospace applications. The applications concerned require reliable operating system support that must accommodate networks of computers. The report addresses the problems of constructing such operating systems, the communications media, reconfiguration, consistency and recovery in a distributed system, and the issues of real-time processing. We include a discussion of suitable theoretical foundations for the use of atomic actions to support fault tolerance and data consistency in real-time object-based systems. In particular, this report addresses:

- Atomic Actions
- Fault-Tolerance
- Operating System Structure
- Program Development
- Reliability and Availability
- Networking Issues

This document reports the status of various experiments designed and conducted to investigate embedded operating system design issues. We describe experiments and measurements of the distributed operating system UNIX United, a system chosen for study because of its use of remote procedure calls and its level structure. In addition, we introduce several concepts which we believe are very important to the economical and efficient design of embedded systems.

To support EOS, our experimental real-time Embedded Operating System design, we are constructing a portable object-based development system called INDEED. INDEED provides an incremental development environment aimed at the particular needs of object-based real-time system construction. EOS is representative of a family of operating system designs based on a General Layered Operating System construction methodology called GLOSS. In addition, we have implemented a portable and reliable compiler for Distributed Path Pascal, the real-time programming language in which we propose to conduct many of the experiments.

1. Project EOS Overview

Since 1979, the *Software Systems Research Group* at the *University of Illinois* has been working with Dr. Edwin C. Foudriat of NASA Langley to develop methods and techniques for the construction of real-time embedded operating systems for aerospace applications. The major practical research contribution produced by this co-operative effort is an experimental real-time programming language called Distributed Path Pascal[Campbell 83]. Distributed Path Pascal incorporates strong-typing and allows object-oriented programming, modularization of code, separate compilation, and fast real-time execution.

Distributed Path Pascal has been the development vehicle used to study many prototype systems and research issues. The group has designed several small operating system components[McKendry et al. 80, Kolstad 83] based on an object-oriented view of a computer system. This view accommodates the design of autonomous operating system components networked together as "remote objects"[Kolstad 83]. The research project has also produced major contributions (some 18 published papers and 28 technical reports) in aspects of system design including protection[McKendry & Campbell 80b], fault-tolerance[Wei & Campbell 80, Schmidt 83, Campbell & Randell 83, Campbell & Anderson 83], fault-tolerance in real-time systems[Horton 79, Wei 81, Leistman 81], atomicity, fault-tolerance, and consistency[Jalote & Campbell 83, Mickunas et al. 84b], and distributed data base consistency[Mickunas & Jalote 83].

Our current research concentrates on applying the results of our previous research to the design and construction of components of a prototype distributed real-time embedded operating system (EOS). The major requirements for EOS are listed below.

Real-Time Response. Components and subsystems of the application must have support to enable them to respond to I/O events in real-time; that is, fast enough to provide control for the physical system in which the computer is embedded.

Reliable Operation and Fault Tolerance. System components may be used to implement critical lifesupport and hardware survival functions, and must have a very low likelyhood of failure. Fault tolerant techniques should be employed to achieve levels of reliability beyond those that can be achieved by conventional software engineering methodologies.

Autonomous Operation. The system should be a dynamically reconfigurable collection of distributed, loosely-coupled, highly autonomous components. Such systems support failure isolation, standby sparing, triple-modular redundancy, and majority voting. The modularization of components improves reliability and facilitates maintenance.

Design and Maintenance Support. The development of an application will consist of the design, construction, configuration, testing, and maintenance of highly autonomous objects and collections of objects. This development process must be supported by appropriate tools and facilities. In particular, these tools must allow fast prototyping, system instrumentation and debugging mechanisms, dynamic upgrading of object implementations, reconfiguration, reusable software components, test-bed validation, performance evaluation and tuning.

This report describes the results of the EOS project for the six months from November 15th through the present. During this time we have:

- completed a Berkeley 4.2 implementation of a portable Distributed Path Pascal compiler and interpreter which uses sockets for inter-machine communications;
- investigated practical designs of existing distributed systems including UNIX United;
- ported UNIX United to Berkeley UNIX;
- made preliminary performance measurements of UNIX United;
- examined networking issues of distributed systems;
- developed programming and fault-tolerant system concepts based on atomic actions;
- designed a development environment for object-oriented systems which aids in the construction of distributed software;
- investigated improvements to Path Pascal's scheduling primitives;
- completed a portable Path Pascal code-generating compiler for UNIX and stand-alone systems which will support production development of Path Pascal programs;

- planned the overall structure of EOS.

In section 2 of this report we describe our work on providing fault tolerance and consistency through atomic actions. Software fault tolerance is still a relatively new area and little attention has been given to the practical and theoretical aspects of supporting fault tolerance in real-time systems.

Section 3 examines networking issues for distributed real-time systems. We include a case study of the distributed system UNIX United[Brownbridge et al. 82] which supports remote procedure calls and remote file access. The results from our study have provided many insights into the the design and structure of distributed operating systems. Improvements made to UNIX United are also described. These improvements have led to substantial savings in time requirements of operations. Discussions of TCP/IP protocols and optical fiber networks are also included in this section.

Section 4 contains a description of INDEED, an INcremental Development Environment for Extensible Distributed systems. INDEED is object-based and provides a means to prototype, extend, debug, instrument, test and maintain embedded systems. The system supports dynamic reconfiguration, remote operations, and distribution on a network of processors.

Section 5 discusses the issues relating to the structure of operating systems. The General Layered Operating System Structure or GLOSS provides a methodology for building a family of operating systems from reusable components. Depending upon the choice and manner in which these components are combined, systems with different properties can be obtained.

Section 6 contains a discussion of several unresolved language design issues related to the problem of specifying reliable real-time systems, including fault tolerance, atomic actions and scheduling. We consider some solutions in the form of system and language primitives for supporting such facilities. Improvements to Path Pascal scheduling primitives are outlined.

In section 7, we describe recent advances in Distributed Path Pascal implementations. New network provisions, utilizing the more general socket mechanism of Berkeley UNIX, have been incorporated into the Distributed Path Pascal Interpreter. The new provisions allow better support for distributed software development. Work is nearing completion on a new production Distributed Path Pascal compiler in which the code generation phase is the same as that used to support the Berkeley version of the UNIX portable C compiler. The front end of this compiler is implemented using an LALR parser and components taken from the Berkeley Pascal compiler. In addition to offering improved performance, the software will be immediately portable to many different machines, and will provide a reliable production environment for both stand-alone and UNIX-based Distributed Path Pascal.

2. Atomicity and Fault-Tolerance

The single most important requirement of a real-time system is that its performance must satisfy timing constraints. However, the reliability of an embedded real-time system is often critical to the success of applications such as missile launching systems, space stations, and flight control systems. The tools and concepts employed in real-time system construction should help the designer achieve both a high degree of reliability and a high level of performance. The use of atomic actions can help attain these goals.

2.1. Atomic Actions

An atomic action is an operation, possibly consisting of many steps performed by many different processors, that appears "primitive" and indivisible to its environment. At some "level of abstraction", the atomic action transforms the system from one state to another with no visible intermediate states. To the environment, the atomic action has the properties of indivisibility, non-interference and strict sequencing. Atomic actions may be nested. By definition, atomicity implies that no communication can occur between processes performing the atomic action and processes outside the atomic action. This restriction is necessary to ensure that the "internal states" of the atomic action are not visible from outside the action, which would destroy the property of indivisibility. Whenever we use "atomic action" in this section, we are referring to a *planned atomic action*[Anderson & Lee 81], that is, an atomic action which has been programmed to occur rather than one which arises by circumstance.

There is another view of atomic actions held by Liskov[Liskov 83] and Davis[Davis 78]. They require that atomic actions should not only be indivisible, but should also be *recoverable*. This means that the effect of an atomic action is "all-or-nothing"; either all the objects remain in their initial state, or all the objects change to their final state. If a failure occurs it must be possible to either complete the action or to restore all objects to their initial states.

We believe that indivisibility is fundamental but recoverability is not. Recoverability is a property which is not needed for all applications. If recoverability is desired, it should be constructed using primitive atomic actions. Moreover, the imposition of a general form of recovery (such as backward error recovery) can result in a loss of performance, which is unacceptable in systems where performance is critical. As Dr. Foudriat points out[Foudriat et al. 84], performance-oriented systems often require minimal synchronization and simple recovery schemes which can be provided by forward error recovery or reinitialization. In such systems, recovery control should be left to the programmer. Furthermore, as pointed out in[LeBlane 84], the "all-or-nothing" definition of atomic actions is not well suited for real-time distributed systems, simply because many operations in such systems do not naturally behave in that way. (For example, consider the effect of an operation changing the control surface of an unstable aircraft.) Our view of atomic actions does not impose recoverability, and yet it provides a structure which can be used by the programmer to easily provide recovery in a distributed system.

In the next two sections we discuss the application of atomic actions to two major reliability problems: data consistency and fault tolerance. We also discuss our contributions to these areas.

2.2. Atomic Actions to Ensure Data Consistency

Each process accessing shared data (for example, configuration tables in a network of machines) does so under the assumption that the consistency constraints on the data are satisfied. However, if many processes access the data in an uncontrolled fashion, the data can become inconsistent. If the shared data becomes inconsistent, processes may make incorrect decisions and perform invalid computations. This is a problem which must be solved satisfactorily in any distributed system that employs shared databases. Although the database requirements of current aerospace applications are modest, future requirements for databases in embedded systems will increase with increases in demand for real-time analysis of large amounts of data. This is particularly relevant to systems that must operate without much support from ground control. Moreover, real-time system mechanisms which ensure database consistency must also provide satisfactory performance and must be able to support time-constrained programs. Examples of database applications which might occur in such real-time systems include surveillance, data collection, inventory, error diagnosis, and knowledge bases for expert systems.

We will use database terminology to formally specify this problem. A database consists of identifiable data items called *entities*. The unit of processing on a database, called a *transaction*, is a sequence of read and write actions on entities of the database.

Even if a read or write on an entity is assured to be atomic, the database may lose its consistency because of concurrent access by different transactions. This is a result of the fact that the transaction is the unit of consistency rather than the individual read or write action This problem was first described by Eswaran, et al.[Eswaran et. al. 76]. The solution to this problem is to insure that each transaction operates as if all other transactions are indivisible operations. To maintain the consistency of the database, a transaction should be an atomic action. In the database literature, the problem of providing atomicity to transactions is often referred to as the "concurrency control problem".

Eswaran, et al. also proposed a locking protocol called the "Phase Locking" protocol, which insures atomicity for each transaction. Many subsequent proposals have been made to provide atomicity in databases. In **Appendix A** we describe a new protocol, the "Re-Read Protocol", for controlling concurrent access to a database. The protocol uses a combination of preventive and corrective measures to maintain consistency, and always grants a Read request without delay. The protocol is deadlock-free, requires no backup data, and supports a greater degree of concurrency than Two Phase Locking. A transaction is never aborted or delayed indefinitely by the protocol.

The guaranteed absence of deadlock, abortion, and indefinite postponement, which this protocol provides, is of particular interest in real-time systems. If transactions can deadlock, as in Two Phase Locking, the system requires mechanisms for deadlock detection and subsequent abortion of one or more transactions. This leads to unpredictable delays and additional overhead, which are not acceptable in real-time systems. The usefulness of the absence of indefinite postponement of transactions needs no elaboration.

2.3. Atomic Actions for Fault Tolerance

As mentioned before, reliability is a major concern in real-time systems. It is very difficult, if not impossible, to write provably correct programs. It is also difficult to demonstrate that a correct program has been translated precisely into correspondingly correct machine code. Fault-tolerance techniques provide the means to keep the system running even if the design of the system has bugs. Fault tolerant techniques enhance system reliability beyond the point which can be achieved by regular software engineering methods.

Some of the major requirements for fault tolerant techniques in real-time systems are listed below.

a) *Both forward and backward error recovery*, should be supported. Forward error recovery is particularly useful in time-critical situations in which the source of the error can be determined. Backward error recovery is required to recover from errors of unknown origin. Both schemes should be provided in a manner in which they may complement one another. Where good performance and high reliability are required, both techniques can then be used.

b) *Uniformity*: Preferably, the system should support both forms of recovery in an uniform fashion. This would permit easy use of the techniques, and result in more readable and reliable programs.

c) *Simplicity*: Fault tolerance techniques should be simple, so that one can easily ascertain that the techniques themselves do not have faults. Since error recovery measures are less frequently exercised in system tests, it is important that the measures are easy to program and understand.

d) *Performance*: The techniques must have good performance in order to be useful in real-time systems. This means that the recovery time and error propagation in the system should be bounded.

e) *Flexibility*: The programmer should be able to select particular recovery techniques to meet a given set of requirements. We doubt that any one scheme of error recovery is sufficient for an embedded real-time system. It would be undesirable to impose a particular scheme, such as the recoverable atomic action, upon the design of such a software system. Instead, we propose that the designer should be able to choose one of several techniques. In this section we will argue that atomic actions, as we define them, provide a structure for recovery within which these requirements can be met.

Fault-tolerant techniques, in contrast to fault avoidance methods, use protective redundancy to ensure that an erroneous system state does not lead to system failure. These methods attempt to place the system in a state from which processing can proceed and failure can be averted. Techniques for fault tolerance are usually classified as backward or forward error recovery techniques. *Backward error recovery* involves backing up one or more processes to a previously checkpointed state, which is expected to be error free, and then attempting to continue further processing. In contrast, *forward error recovery* aims to identify the fault and correct the erroneous state of the system before proceeding with normal processing.

Both of these fault tolerance techniques have four major phases[Randell et al. 78]: error detection, damage assessment, error recovery, and fault treatment/continued system service. *Error detection* by software is done by checks on the state of the system which are usually performed just before leaving the system (or sub-system). The checks should be derived from the specifications. If the hardware detects errors, the system is usually informed of the error by a hardware interrupt.

Atomic actions provide a convenient structure to support damage assessment and recovery. In the atomic action framework, the damage due to a fault is confined to some atomic action which contains both the fault and the detection of the error resulting from the fault. Since a fault can only be detected by detecting its manifestation, that is, an erroneous state, it may be difficult to pinpoint the fault and consequently the extent of the damage due to the fault. The initial estimate is that the damage is confined to the deepest nested atomic action enclosing the error detection point. If the recovery in this atomic action does not succeed, then recovery is attempted in the next enclosing atomic action. This process continues. The nesting property of atomic actions permits this approach for damage assessment.

The atomic action containing the damage can be inspected to determine the cause of the error, as a forward recovery technique would do. Alternatively, all the computation performed inside the atomic action can be regarded as being suspect. In this case, the computation should be discarded and the system returned to the state it was in at the beginning of the atomic action. This is the backward error recovery approach; the state of each process must be saved before that process enters the atomic action, in order to provide an easy way to roll back to a previous consistent state. For both kinds of recovery, atomic actions provide bounds on the damage produced by the fault. In the case of backward recovery, planned atomic actions also prevent the *domino effect*. The domino effect is particularly undesirable in real-time systems because it makes it difficult to bound the amount of recovery needed and consequently adds uncertainity about the recovery time.

The use of atomic actions to support backward recovery in concurrent systems was first proposed by Randell[Randell 75]. The construct proposed is called a *conversation*. This scheme uses the static definition of the boundary of an atomic action to define a priori the limits for error containment. The use of atomic actions for forward recovery is proposed and discussed by Campbell and Randell[Campbell & Randell 83]. It has been suggested that the two techniques of providing fault tolerance should be used in a complimentary manner[Randell et al. 78, Cristian 82]. Cristian first proposed a scheme to integrate the two techniques by considering backward recovery to be an exception handler for a *failure exception*. The scheme has been extended to asynchronous systems in[Campbell & Randell 83] employing atomic actions. This proposal is described in **Appendix B**.

Few implementations permit both approaches to be combined within a particular application. Even fewer techniques are available for the construction of fault-tolerant software in systems of concurrent processes and/or multiple processors. **Appendix C** contains a proposal for supporting forward and backward error recovery in a system of Communicating Sequential Processes (CSP). The proposal uses atomic actions, called S-Conversations, to support the different recovery schemes. The S-Conversation is implemented using CSP primitives. Syntax and consistency are checked during compilation and at run-time. The S-Conversation uses a small set of primitives to uniformly provide both forward and the backward recovery.

2.4. Future Work

We have shown how we can use atomic actions to preserve data consistency and support fault tolerant techniques, without significant sacrifices of performance. However, the application of atomic actions is not limited to these areas, but can aid in improving software reliability in many additional ways. Our preliminary investigations have revealed that atomic actions can be used to develop techniques to structure and design concurrent systems. They can also simplify the problem of proving parallel programs correct. We are currently investigating these areas. In **Appendix D** we describe some of our preliminary findings.

3. Networking Issues

Future real-time embedded systems will be designed as networks of many different specialized processors. Such networks offer many advantages over a centralized processor. By decomposing the control of an embedded system into many independent control systems, many of the problems of real-time processing can be eliminated. Networks provide reliability by physically separating different functions of a system, reducing the possibility of error propagation, permitting replication of important components, and reducing the risk that a component failure will result in the failure of the whole system. However, the structure of a reliable software system for a network of processors is an active topic of research.

In this section we discuss a distributed operating system, called UNIX United, which we have studied in some depth. We also discuss some networking issues which arise in optical fiber networks. We studied UNIX United because it is simple, available, portable and includes many desirable features. UNIX United was originally designed to connect an arbitrary number of Version 7 UNIX systems into a distributed system which has the same properties as a single processor UNIX system. As such, UNIX United provides an easy-to-use workbench for constructing EOS. However, the major contribution to EOS results from the study of the principles that underly UNIX United's construction. UNIX United is a UNIX® extension which provides transparent access to remote resources and files, communications between remote processes, and process migration.

3.1. UNIX United

The Newcastle Connection is a software package that implements UNIX United by enhancing UNIX with mechanisms for transparent access to remote files, invocation of remote processes and procedures, and communication between processes on different processors. Despite the additional capabilities, the UNIX United that results from combining UNIX systems is identical to a single processor UNIX system at the application program and user interface level. Thus, software that is prepared to run on a UNIX system can easily be reconfigured to make use of the distributed processing facilities of UNIX United. Similarly, users may invoke the distributed facilities of the system using standard UNIX commands.

The Newcastle Connection is implemented as a set of subroutines which are linked into the user program. These subroutines replace the standard entry points for system calls, the mechanism whereby the user program communicates with the UNIX kernel. These subroutines determine whether a requested action should be performed locally or remotely, and perform the appropriate packaging, message transmission, and interpretation of results.

UNIX United involves no modifications to the UNIX kernel. This allows it to be moved easily between different implementations of UNIX. Users uninterested in the extensions provided by UNIX United can link their programs with the default subroutine libraries and avoid all overhead associated with the UNIX United system call mapping.

3.1.1. Properties of UNIX United

Level Structuring : UNIX United is modular and small because it forms a single, well-defined layer within the system. The interface between the Newcastle Connection and the UNIX kernel is, to all intents and purposes, identical to the interface between application programs and the Newcastle Connection. Other layers can be added (and have been added) between user processes and the kernel without requiring changes to the kernel or the Connection. These benefits are identical to those which we proposed would come from using the execute statement [McKendry & Campbell 80a] to build level-structured operating system.

Remote Procedure Calls : Remote procedure calls provide a basis for implementing all distributed services and functions. The scheme adopted is similar to that proposed to support remote objects in Path Pascal except that it is not object-oriented. In particular, the performance measurements of UNIX United show this to be an effective and efficient way to provide general remote access to resources.

Efficient Remote File Access : An efficient remote file access should allow permissions and accessing methods to be set up at an "open file" request rather than with each read and write request. UNIX United implements this scheme which allows many performance optimizations to be made.

Variable-Length Datagram Service : There are several advantages to the Newcastle Connection scheme of providing a network protocol based on a variable-length datagram. First, the datagram can be transmitted quickly using a scatter/gather packet transmission scheme. Second, large and small datagrams can be sent by using different protocols; for example, long datagrams may use protocols adapted from file transfer protocols. Future releases of the Newcastle Connection will include "adapter" software which will provide routing and protocol selection based on the length of the data to be transmitted.

Hierarchical Naming Scheme : The hierarchical naming scheme used by the Newcastle Connection is a very simple and effective way of naming resources inside a network. The scheme eliminates the need for name servers within the network. This improves reliability and efficiency since the resources in the system can be named in a completely distributed manner. However, it would seem appropriate to extend the naming scheme to allow the specification of physical names, such as remote process id, remote file descriptor, and remote user group. In addition, for consistency purposes, it would appear very desirable to be able to use path names that start from the root of the hierarchy, as well as path names that start at the "current working directory" or at the "root" of the local host.

Process Mapping : UNIX United allows the environment of a process to be mapped from one machine to another. Thus open files, parent and sibling processes, and other process attributes are independent of the machine on which the process executes. This has many advantages including generality. It actually simplifies the network support structure provided by the Newcastle Connection, and allows the code to be very small and efficient.

3.1.2. Experiments on UNIX United

We have made several performance measurements of UNIX United running on departmental equipment. Two VAX 11/750s and two 80 megabyte CDC permanent-surface disk drives were used for the experiments. The machines were connected by a 10M Ethernet®.

Remote procedure calls can provide most network facilities efficiently. In one comparison, we transmitted 2 megabytes of data from a local disk to a remote disk using UNIX United's copy command which is implemented by remote procedure calls. The transmission time took 2.1 minutes as compared to a local copy time of 1.75 minutes. Remote random access to a disk took 26 milliseconds per byte/record. Details of these measurements can be found in **Appendix E.**

The largest part of the time required for the file copy was a result of disk latency and arm contention; network transfer times were a small part of the overhead. We believe the next largest overhead in remote access was caused because the Newcastle Connection remote procedure call transmits data as a sequence of packets. Stream protocols, which permit transmission of several packets at a time using a "sliding window" or similar protocol, are much more efficient. The Berkeley UNIX "rcp" (remote copy) facility employs such a protocol. In the latest release of UNIX United, some of this overhead has been reduced by a new protocol which implements scatter/gather transmission for remote procedure calls with large amounts of data. However, as of the time of writing this report, we have yet to measure the effect of this new protocol on performance.

Access to open files can be optimized by utilizing a buffer cache, as done in UNIX. Such a technique can greatly improve performance if the program that accesses a file exhibits locality of reference in the records selected for read or write. The cache can also improve performance when several processes are accessing the same open file. It is not possible to cache a remote file in a local host if processes in multiple hosts have opened the same file for reading and writing. In one experiment called *iotest*, described in **Appendix E**, we measured some of the effects of not being able to provide a local cache for remote file access.

3.1.3. Lessons Learned

Many lessons were learned from the experiments we performed. We expect them to be of great help in the design of EOS. We found that it is practical to permit remote access to records of a file rather than to require the transmission of the whole file from one system to another. In a system in which storage space is at a premium, the ability to have processors select just the information they need from a remote resource is a great economy. The performance of UNIX United when used for single record access justifies this approach.

We found that a distributed system must be carefully tuned. Initial timings of UNIX United revealed many mismatches between packet size, remote service request sizes, and buffering sizes. Typically, most remote procedure calls were only a few bytes in length, while the maximum packet size supported on our Ethernet is 2 Kbytes. By using the packet size of 2 Kbytes, the performance improved by a factor of three.

The use of *light-weight protocols* for remote procedure calls have significant performance advantages. The performance of UNIX United was very sensitive to the amount of copying occurring in the protocol handlers. By improving the copying algorithms, we improved the performance by a factor of five. For EOS, we propose protocols which eliminate copying. We also propose that for unavoidable copying, we use subroutines which take advantage of the hardware architecture.

All components of a system interface must be *consistently networked*. For example, we found that the mapping of signals and exceptions between remote processes was very convenient. However, such mappings should also be implemented consistently. One of the bugs we discovered in UNIX United was that signals were mapped in a somewhat ad hoc way for certain special process configurations. When a process is executed remotely, a stub process is left to intercept signals from the user. In Version 7 UNIX, this proved to be satisfactory. However, in Berkeley UNIX there are several additional signals including one to suspend a running process with the intention of being able to restart that process at a later time. This signal and the "kill" (abort) signal cannot be intercepted by a UNIX process. The effect of sending either of these two signals to a remote process is to suspend or kill the local stub process instead. The bug-fix involves making sure that signals are mapped correctly within the Newcastle Connection layer instead of allowing them to be directly intercepted by the stub processes.

In UNIX, signals are used to communicate exception messages. In EOS, this problem corresponds to ensuring that exception conditions that are sent to a remote object are reported to the actual object, and not intercepted by a run-time mechanism supporting the distribution of objects. The exception mechanism required for the run-time mechanism and network conditions should be a implementation concern completely separate from that of the application exception handling scheme. The exception handling routines of the run-time and network support can, of course, signal exceptions to the application.

The disk block size used by utilities for portable operating systems should be *machine independent*. For example, the Berkeley UNIX 4.1 stdio library utilities assume that it is best to copy 1Kbytes blocks of data from one file to another. In the remote file access case, the copy utility actually resulted in inefficient use of the network packet size. It is worth noting that in the new release of Berkeley UNIX (4.2), the stdio library utilities now query their environment as to the appropriate block size to be used.

3.2. Networking Mechanisms

We have investigated several protocols to support networking for EOS. In order to provide maximum communication compatibility with other systems, the networking mechanisms should be implemented using standard protocols. However, our experience from UNIX United suggests that several existing ISO Open Systems Interconnection Model protocols[Zimmermann 80], including TCP/IP[Postel 81a, Postel 81b], are inefficient. Our current plan is to investigate some of the new OSI protocols which may support light-weight remote procedure calls more efficiently. In addition, we understand that many of the less efficient standard protocols are being embedded within single chip components which may result in improved performance.

Most of the UI Department of Computer Science computers, specifically two VAX 11/780s, seven VAX 11/750s, one Pyramid, and ten SUNs, communicate over a 10 Mhz Ethernet via the TCP/IP protocols. These protocols offer reasonable reliability but make inefficient use of the host computer if they are implemented in software. The internet addressing scheme is becoming very popular and would allow any EOS system to coexist with other networked systems. In general, we believe that it is better to provide a general interface to the network in EOS rather than make EOS depend too heavily upon one form of networking software support. Our work on Distributed Path Pascal provides such an interface, based on variable length messages containing remote procedure calls transmitted using UNIX pipe-like operations. We have found that it is very easy to port Distributed Path Pascal from pipes to sockets to datagrams. The choice of network protocols for EOS seems to be primarily an optimization question.

3.3. Optic Fiber Networks

Much of our research on distributed systems has used Ethernet for prototype experiments. However, such a networking media may not be adequate for major embedded distributed system applications. Project EOS has also examined several other networks in cooperation with other projects in the Department of Computer Science at the University of Illinois. In particular, we have compared ring networks and Ethernet-like networks based on optic fiber transmission. A two megahertz optic fiber ring network has been constructed in the Department and has been interfaced to two VAX 750s running Berkeley 4.2 UNIX. This network is capable of being upgraded to speeds in excess of 100 megahertz. Currently, a 30 megahertz version of the network is under construction. In our opinion, networks that provide transmission rates in the same order as I/O devices are not only feasible but very desirable, especially in applications requiring responsive real-time systems.

3.4. Some Advantages of Optic Fiber

Optic fibers have several advantages over other communication media particularly in aerospace applications:

● They are less susceptible to environmental disturbances. For example, they are immune to electromagnetic interference, lightning, static, radio frequency interference, ground loop problems, and solar flares.

● They do not generate any environmental concerns; for example, they do not generate electromagnetic interference or radio frequency interference.

● They provide a secure transmission service which is impossible to monitor, and they prevent crosstalk between different communication circuits.

● Current optic fiber technology now provides a low loss transmission medium which can be used over large distances, requires few repeaters and is more reliable than previous mediums. Although initial space station systems are likely to be small, some future space platforms may require distributed computing support that extends over a considerable distance.

● The physical properties of optic fiber also make it a very attractive medium to be used in aerospace. It is lightweight, flexible, and small in size.

For these reasons, despite its current expense, optic fiber has become one of the most attractive communications media for future embedded systems.

3.4.1. Ethernet with Optical Fibers

The Ethernet protocol permits several networked computers to compete for access to the Ethernet, a base-band transmission on a coaxial cable. This protocol has been also implemented on optic fiber, for example by Ungermann Bass Inc. The protocol introduces random delays into the transmission of messages but has become very popular for commercial use because of its availability and ease of use. A major disadvantage of the protocol is noticed in high-transmission rate networks in which the nodes are geographically distributed. (For 10 megahertz the limit is about a mile and a half.) In such cases, the propagation delay of the signal along the cable may be so large as to prevent the collision detection mechanism from functioning reliably. An Ethernet tends to become saturated at high-levels of packet transmission. When used in combination with optic fiber, the Ethernet protocol may be implemented with a transmit and receive fiber. Communication is achieved by transmitting a packet of information from a node along the transmit fiber. All of the transmit fibers meet at a central node where a transducer and amplifier retransmits the packet back out on all the receive fibers. The central node also detects collisions by noticing simultaneous transmissions on the transmit fibers.

3.5. Illinet

Illinet[Cheng et. al. 80] is an example of a token ring network and is similar to the "Distributed Computing System" at the University of California, Irvine. Illinet transmits information at a rate much faster than that of the Ethernet because collisions are avoided by passing a token around the ring. Current work is constructing an Illinet implementation that supports a link bandwidth of 32 Megabits per second. ECL circuits are used to interface the optical transducers to a communications processor.

To achieve a reasonable speed, most of the network access and link control protocol function is implemented in hardware. The link control protocol enables nearly all of the 32 Megabits to be available for interprocessor communication. The network is organized as a ring with a relatively short loop delay around which packets are transmitted. The network control structure is distributed and is based on a token control scheme. This scheme makes message transmission times deterministic which is a desirable property in real-time network applications. A node which receives a token may send a packet. If the node does not wish to communicate or has sent a packet, the token is passed on to the next node in the ring. A recovery protocol is used to ensure continued service if the token is lost.

The hardware includes an associative addressing scheme that permits broadcasts to all or selected network nodes. Each packet includes a sixteen bit destination address. This destination address is a logical address in that it is independent of the actual physical node which may receive the packet. Each node may load a table of destination addresses for which it wishes to receive packets. As a packet passes a network node, the hardware compares the packet destination address with the table and and generates an acknowledgement if a match is obtained. Several nodes may acknowledge a packet. The scheme permits a network resource, associated with a particular network address, to be transferred transparently from one network node to another without any control mechanism being involved. The address recognition hardware and link control protocols enable efficient broadcast communication. Illinet is not limited in length and simulations of the network (using Path Pascal) have shown that it does not saturate but gracefully degrades as network traffic increases. Illinet is interfaced to network nodes by a programmable DMA interface. It is currently being used with the standard TCP/IP protocols and has proved very reliable.

Illinet does not require a central controller for clock synchronization or access control. Segments of the ring are joined at nodes by a ring adapter. Each ring adapter acts as a repeater by rebroadcasting the incoming data stream data. The communications processor breaks messages into packets and sends these packets to the ring adapter output buffers. From there they are transmitted one at a time each time the adapter receives a token. After transmitting one packet, the adapter waits to receive the acknowledge field of that packet once it has traveled around the ring. The data packet is retransmitted until a positive acknowledgement is received from all the nodes that are receiving packets with the given packet destination address. The token is then passed on to the next node in the network. The processor receives messages from the host by DMA transfer. Receipt is acknowledged by an interrupt sent to the host processor. The host then signals "commence sending" and the processor transmits the message. Packets received from the network are acknowledged and assembled within the ring adapter and processor. A packet is not acknowledged if the CRC check fails or there are not enough free buffers within the processor. When a message is completed it is transferred by DMA to the host and the host is interrupted. Hosts must assure reliable message sequencing.

4. INDEED: An Incremental Development Environment

The construction of reliable, embedded real-time application software is an enormous task demanding tools of much greater flexibility and power than required for the development of conventional systems. The size, complexity, and special critical requirements of the computational tasks of space system software require sophisticated approaches to development methodology including support for reliability, portability, fast prototyping, programming-in-the-large, and incremental development. The choice of object-based programming to achieve some of these requirements imposes the additional requirements of protected support for dynamic allocation, revocation, and reconfiguration of separately developed, highly autonomous components[Foudriat et al. 84]. These components may reside together on a single machine, or may be distributed over a possibly non-homogeneous network of autonomous computers and devices. Policy decisions regarding placement of an object on a particular hardware device or at particular physical location on the network should be independent of the specification of the object, and should not obstruct dynamic reconfiguration. Where special purpose hardware requires that a specific set of services be identified with a particular hardware node, this should not be visible to client-objects requesting the location-dependent operations. This allows hardware reconfiguration without requiring revision of existing software implementing the client objects, that is, configuration portability.

4.1. Extensibility and Incremental Development

The operating system to support advanced object-based applications must provide efficient and reliable mechanisms for inter-object exchange of services. The primary task of the object-based operating system is no longer limited to providing a fixed and immutable set of system services to a series of highly isolated tasks. For very specialized applications under severely critical requirements, such as those of real-time space station software, the conventional virtual machine (such as the Path Pascal or ADA VM), or some component providing a sub-set of the virtual machine system services, may prove inadequate, inefficient, or too restrictive. Such system services can be provided by new server objects within the system or application, as long as these server objects are available within the environments of the intended clients. For example, since objects themselves provide "persistent data", but allow specialized and protected operations on that data, the classical OS file system services can all be provided by classes of "file objects" with appropriate operations[Appelbe & Ravn 84]. Rather than define and provide all external services required by application software, the role of the object-based operating system is to enable objects to provide (use) services to (of) other objects in a reliable and protected manner.

Thus the primary responsibility of the object-based operating system is to create, maintain, reconfigure, and monitor access paths among sets of highly autonomous objects and nodes (collections of objects), and to protect against access other than that defined and allowed by the operations.

In a system designed to support the exchange of services between highly autonomous components, communication between client and server objects must be efficient and well-defined. In addition, these communication mechanisms must provide for dynamic reconfiguration of access paths during initial prototyping, during incremental development, and during system debugging or reorganization. At a given stage in the development of an application, and at any instant during the operation of the system, there exists a collection of objects, ranging from finished implementations to prototypes or even stubs, which exchange services in the form of operations. If one object has the capability to request a service from another object, then we say that there exists an access path from the client object to the server object. Note that this does not imply that the client **must** request the service, but only that the client **may** do so. This collection of all existing objects, along with all access paths between pairs of objects, forms a general graph structure, which is maintained by the operating system mechanisms. Further restrictions on this structure constitute the system's access control policy, or protection policy.

4.2. Overview of Thesis Proposal for INDEED

INDEED is an incremental development environment which supports object-based programming, environment definition and management, dynamic reconfiguration of collections of compiled objects, and extension of system virtual machines, based on the object model of the systems programming language Distributed Path Pascal[Kolstad 83]. In the INDEED system, the individual components are objects, or highly autonomous collections of objects (nodes). Objects are the smallest units of separate compilation, and may be added, replaced, or removed from a running system in a single atomic action. During development, individual objects may be tested and debugged even though other objects, whose services are required of the object undergoing testing, may not have been implemented or may still be in the prototype stage. Restrictions on allowable access paths are imposed automatically and are enforced by mechanisms within INDEED. By default, these restrictions are the same as those imposed by the scope and parameter-passing semantics of the system programming language. However, other subsystems with specialized access control management are also supported (such as GLOSS layers). For example, for the purposes of debugging, system monitoring, or reconfiguration, full control of the access policy could be granted, which would permit unrestricted access to objects and processes in the system. This would allow manual creation or manipulation of any portion of the potentially fully-connected access graph. It would not, however, allow any operations on an object inconsistent with that object's definition.

INDEED utilizes the Path Pascal language model as the operating system language, in the sense that "system calls" and commands consist of calls on the operations of system objects. Three primitive system object types form the "nugget"[Joy 84] of the INDEED system, the *symbol table object* (STO), the *object compilation manager* (OCM), and *heap management object* (HMO). Firstly, the symbol table object, STO, provides the mapping between source program names of Distributed Path Pascal objects and capabilities to specific instances of those objects and their type definitions. With or without the operations for creating new definitions of object names, the STOs implement environments for all objects in INDEED by viewing and managing certain objects as tables of capabilities. Secondly, the object compilation manager, or OCM, provides traditional DPP compilation facilities, traditional except that

- external references are resolved by calls to the current environment STO, and that
- the unit of compilation is the object rather than the program.

The OCM generates code for object access in the form of indexing through environments, and provides protection by restricting access to legitimate operations only. Lastly, the heap manager object, or HMO, manages memory allocation for all INDEED objects. Thus the HMO views each object as a segment of contiguous locations in "capability-space", which may be either physical memory or virtual memory depending upon the implementation and underlying hardware. Various versions of all three primitive objects can be configured together to form specialized subsystems. These primitive objects are described in more detail below.

In much the same way that the system-wide notion of "file" provides a unifying concept in the UNIX® operating system, the system-wide notion of "object" provides the unifying theme for all structures in INDEED. While the UNIX® file philosophy provides great flexibility and encourages the construction of larger tools from smaller ones, it does so at the expense of the security and efficiency provided by information-rich type checking. The unstructured nature of the UNIX® file implies that full responsibility for checking lies with the user of the file. Thus the programmer using the file data type must include additional, possibly complex, integrity-checking code to make his program robust, leading to inefficiencies and possible sources of error not present in a strongly typed environment. Even more significant is the fact that the original programmer of a particular file application has no guarantee that users of these special purpose files will perform only valid operations on those files, hence there is no system support for the integrity of the data. In INDEED, processes acting on Distributed Path Pascal objects may perform only the legitimate operations associated with those components.

4.3. Environment Management in INDEED

The run-time symbol table managers in INDEED monitor and manage access to all object instances in the flat object-space. Each manager controls access to a disjoint set of objects located in one or more contiguous chunks of (possibly virtual) memory on a given node. This access control is capability-based. The symbol manager provides operations for mapping names to capabilities, returning attributes of objects associated with names (type descriptors), redefining the mapping (new object, new type, new operations), mapping local calls to remote objects via the remote procedure call mechanism, and providing software trapping on operation calls. Just as the UNIX® directory structure permits construction of a hierarchy of files on top of the flat file structure[Thompson 78], hierarchies of objects can be built by creating capabilities to symbol managers within other symbol managers.

The INDEED STO thus defines and manages a hierarchical run-time environment according to the given policy for access control. A subset of the operations defined on the STO object includes exactly those operations provided by the Path Pascal compiler symbol table. These access control policy semantics are equivalent to the rules governing naming in the Path Pascal source language. This is the default access policy. Subsystems of specialized STOs can be built which enforce alternate schemes, or more restrictive schemes, which will allow construction of interfaces between INDEED and subsystems supporting other languages, such as Ada®, HAL/S, etc. By swapping capabilities, the STO can dynamically redefine object implementations for such purposes as modifying and improving existing services, providing stubs for unimplemented objects, monitoring and/or trapping operation invocations on a particular object, and dynamically switching local service calls to remote server objects by switching to Remote Procedure Call filters[Spector 82, Shrivastva & Panzieri 82, Birrell & Nelson 84]. Roll-back of recovery objects[Schmidt 83] can be implemented in INDEED in exactly this manner; the previous state of an object can be recorded by copying the object into a hidden object, perhaps on stable storage, and roll-back would involve swapping the capability for the active object with that of the hidden copy. All of these modifications to the access graph can occur dynamically without the necessity of "bringing the system down" for reconfiguration.

4.4. Storage Management in INDEED

The feasibility of a heap-based system such as INDEED depends heavily on the efficiency with which objects may be allocated dynamically. In INDEED, activation records (stack frames) themselves are treated as (environment) objects and are allocated from a heap. All requests to instanciate objects, including new activation records, are requests made by a single process for an initially unshared portion of free space. If a single global heap (per machine) were accessed by many concurrent processes each time a procedure call occurred, contention and additional overhead during the calls could severely reduce response time.

In order to avoid this added complexity and overhead, the HMO in INDEED manages one or more regions of contiguous memory locations of a "capability space". All of these heap regions are disjoint, and each region is managed by at most one HMO. When a new process is called, the process may be granted a contiguous segment of memory for use as a stack for efficient creation of activation records during procedure (or function) calls. If and when this region is exhausted due to repeated procedure calls, a stack overflow is detected and results in a call to the controlling HMO, which allocates an additional stack-region for the process.

The size of the region granted has great influence on the expected time to the next stack-overflow, and tuning the HMO allocations to the characteristics of a given process allows system performance to be optimized. Note that the stack allocation policy of the current implementation of Distributed Path Pascal is a degenerate case of this more general approach, specifically the case of an initial allocation of the process' estimated stack size, with program termination if stack overflow occurs. Of course, an HMO could refuse to grant additional stack regions to a process, and instead allocate the AR object directly from either the remaining free space or from "holes" in existing stack regions. In this case, additional stack-overflow faults will occur as soon as the process requires more memory. The generalized heap allocation scheme allows tradeoffs between efficiency and space utilization, and allows fine-tuning on the per-process level. When the number of active references to a given object drops to zero, the space allocated to the object can be recovered and added to the free list. Recovery of heap space is the responsibility of the HMO, and is governed by the particular policy in force.

4.5. Incremental Development in INDEED

The third essential primitive component (also an object) in INDEED in the object compilation manager or OCM. The operations provided by the OCM permit both the generation of compiled code for a new object, and the instanciation of that object within the current environment. The OCM differentiates between the use of a capability for environment object indirection and the use of a capability for object operations in the code produced to exercise the capability. In the former case, the object addressed by the implicit capability contains a jump table for operations in objects and access to implementations for all entities visible (defined) in that environment. In the latter case, the OCM generates indexing off the object capability according to the type definition of the object. In this way objects in INDEED may have dual roles as both environment (package) objects and as abstract data type (class) objects.

This use of indirection through environment objects provides the flexibility needed for incremental development, dynamic reconfiguration, multiple implementations (versions), and programming stubs. To ensure that new configurations preserve the integrity of previously compiled objects, we define an object uniquely within a given environment by its index into that environment. If previously generated code for client processes and objects utilizes a given service object, we say that the definition is "active". Hence the correctness of a configuration (access graph) of active objects requires that the definitions preserve the specification semantics of each active object. An inconsistent definition, which would change the object's external interface (set of operations), cannot be installed at an index which is still active. On the other hand, the implementation of a given object, including the code for the operations, the local variables, and the set of environments in which the object is defined, can all be revised dynamically (at run-time) without perturbing the active objects, as long as this revision occurs as an atomic action. This restricts dynamic reconfigurations and reimplementations to those which do not violate the policy mechanisms enforced by the STO, and so preserves the system integrity.

4.6. Production Systems

Various components developed under INDEED can be selected to compose the final production embedded real-time system. Objects in the original configuration which were included solely to provide specialized development services, such as instrumentation, filtering, editing, or simulation, can be omitted in the production configuration without requiring revision to any of the objects selected. Even INDEED primitive objects may be omitted; for example, in an embedded system which does not require further incremental development, the OCM can be omitted. More often, a development-oriented object will be replaced by a specialized version which provides only a subset of the original operations, or which provides restricted operations. For example, one could provide a restricted STO which does not include the operation for creating new object definitions in the environment. The final configuration need include only those objects actually essential to the production system, and can include restricted versions of objects which provide only essential operations. This ability to streamline and tune configurations is critical to efficient real-time performance.

5. Operating System Structure

GLOSS is a methodology for assembling operating system components to provide varying environment and performance characteristics. Such environments may provide debugging capability, software fault-tolerance, triple-modular hardware redundancy, atomicity, real-time response, and transparent access to remote resources. For example, time-critical processes can run in a real-time environment while other computational processes can run in a batch-oriented environment. Machine dependent processes can access particular features of a processor while other processes may be isolated by a virtual address space. Distributed processes can utilize the layers providing transparent access to remote resources while others are isolated on a particular host. All these forms of processes can coexist on the same node and may communicate through common mechanisms. Although the GLOSS methodology is independent of any particular implementation mechanism, the INDEED system is capable of providing the layered environment structure to support this technique.

A system structured according to the GLOSS methodology has a minimal kernel providing a set of basic functions, such as inter-process communication, memory management, scheduling, and perhaps portions of a filesystem.[1] Virtual machines are layered above this basic unit as desired to refine the basic operations of the kernel without altering their function (such as providing real-time environments, distributed filesystems, atomic actions, and fault tolerance). Each layer conforms to the standard interface and forms an encapsulation of the system This serves to isolate and identify related mechanisms that support a particular extension to the basic kernel. This encapsulation may be realized solely by software support or it may be realized by hardware mechanisms such as the "supervisor state".

Processes are allowed to change environments with little or no overhead. However, changing the environment may involve some loss of environment-specific facilities. For example, leaving a distributed filesystem environment will prevent access to any open remote files.

Bill Joy of Sun Microsystems calls this basic kernel a "nugget", a unit smaller than the conventional "kernel"[Joy 84].

5.1. Properties of GLOSS

In GLOSS, the definition of the interface between layers is standardized for a given family of operating systems. This allows different layers to be combined together in different combinations. Communication through this interface is constrained to follow a common protocol within that family of systems. Certain layers may require the inclusion of other layers in a particular order because of the extensions they provide. For example, a TMR layer[Brownbridge et al. 82], which executes a program in parallel on several hosts, may require a distributed filesystem layer to manage access to any replicated files. Each layer is analogous to a "UNIX" filter, intercepting the standard operations requested by application programs and processes in higher layers and mapping them into operations on more primitive layers. For example, a distributed filesystem layer may convert a read file request into a series of lower level requests to transmit a message to a remote host requesting a remote file access.

GLOSS requires that the interfaces between layers be identical. This permits layers to be easily assembled into new permutations. UNIX pipes are a good example of how a standard interface provides great flexibility by permitting elements to be combined into pipelines. All the standard read/write operations function identically on both files and pipes, which allows arbitrary numbers of filters (processes with their inputs and outputs connected together) to be combined by pipes to process information.

Each layer must provide at least a minimal service for every request defined by the standard interface. For example, a Two-Phase protocol "commit" primitive that might be used to implement an atomic action would be defined in an atomic action layer. The atomic action layer may not necessarily be an adjacent layer to the software in which atomic actions request the primitive; there could be several intervening layers which would allow the request to pass transparently down to the atomic action layer. For example, a distributed filesystem layer could simply pass requests for the "commit" primitive down to lower layers.

Layers should implement a related set of system services and should be designed to be "thin": each level should include only a minimum of complexity to provide a basic set of services and no more.

Examples include distributed filesystems (that route standard file-access requests to the appropriate machine), atomic action layers (that isolate the effects of operations within the atomic action until the action is completed), and fault-tolerant layers (that provide error detection and redundant data/computation capability).

Use of layering in this fashion has several advantages. It allows programs and layers to *run in different environments* without needing any change. For example, a program can be debugged in a layer which provides debugging facilities and then "migrated" to another environment without the debugging layer.

Multiple environments can run on the same machine. Several highly-reliable processes may be using the fault-tolerant layers while trusted real-time processes may be running with a minimum of layers and overhead directly interacting with the lowest level kernel.

Process switching is simplified. A process can switch between layers at will by invoking appropriate routines. Sometimes this will involve the revocation of resources. If a process running in a distributed filesystem environment calls a routine in the basic kernel, the process may have to re-open any previously open remote files.

5.2. Mechanisms to Support GLOSS

GLOSS specifies a uniform interface between adjacent layers. The interface would be independent of programming languages, host machines, and operating systems. This would allow GLOSS to provide support for other languages in addition to Path Pascal, such as Ada[Ada Reference Manual]. Distributed Path Pascal would be linked to a GLOSS interface by a small run-time library or by the STO of INDEED. For EOS, the GLOSS interface will include embedded operating system run-time support services. These services would define support for fault-tolerance, real-time and deadline scheduling, protection, and networking primitives.

Capability managers, implemented as STOs in INDEED, provide a basis for a GLOSS implementation. Each capability provides protected access to a fixed set of services. Each layer has its own set of capability managers, and provides the same service interface. A particular application will be bound to the actual services via its set of capabilities as determined by the capability manager. For efficient real-time applications and layers, the capability manager may be eliminated by replacing the capabilities with statically bound pointers. At lower levels in the system, some services may be undefined. The default stub objects in INDEED can be used to trap such undefined calls, or to perform a null default service operation that returns an "unsupported operation" error.

Many current computers provide several processor operation "modes". The VAX and PDP-11 families provide "kernel", "executive", "Supervisor", and "User" modes. The Motorola 68000 family architectures only provide 2 levels of privilege. These modes can be used to separate and protect the name and address spaces of various layers.

5.3. Naming Layers

The system should support a name mapping mechanism within the network. A naming layer is used to map a "transparent name", i.e. one that is host dependent, to a physical address. This physical address could be anything from a UNIX United path name to a specific byte on a disk. Naming layers also support file replication, which is important for implementing fault tolerance. Should a particular host of a replicated file become unavailable, the layer should map the file name to another other host which has a copy. One implementation of a naming layer is map name service requests to invocations on a name server object on the network. A name server would map the name to a particular host and file system entry. Another implementation of a naming layer is the Newcastle Connection. It is important that the means of mapping the name is transparent to programs above the interface of the naming layer.

The naming layer of a reliable distributed operating system provides critical services. Should the service not function properly, then each network node could be isolated. For this reason, measures must be taken to ensure the continuity and integrity of the name layer. It is good practice to distribute the responsibilities of name service to several nodes in the network. Thus, if one node were to crash, the entire system would not be lost.

5.4. Distributed Kernels

From our experience with UNIX United, we believe that a distributed operating system would do well to put some of the remote access routines inside the kernel. If the "distribution layer" of a distributed operating system resides in user space (as the Newcastle Connection does in its current form), several problems occur. The first problem is that it is much costlier to execute in user space rather than kernel space. A large increase in speed can be achieved if the network routines are moved into a "trusted" layer where they can share critical network and process identity information reliably. This also reduces the size of the user programs. A second problem concerns the handling of exception conditions. The exceptions required to implement the abnormal conditions concerned with networking support are independent from the exceptions required in the domain of user applications. When there is no clear distinction between user applications and networking provisions, exceptions (signals in UNIX) may be handled inappropriately (as in the kill and suspend signals sent to stub processes in UNIX United).

6. Language Design Research

This year we have studied methodologies and mechanisms for supporting distributed multiprocessor computation and real-time programming. We have investigated the problems of scheduling access to resources and of providing atomic actions and fault-tolerance in such an environment. We have considered providing both system primitives and programming language mechanisms to support such facilities. In this section, we outline some of the results of our study of scheduling issues. Because of certain implementation issues, we believe it may be more convenient to implement scheduling facilities as extensions to a programming language rather than as operating system primitives.

Path Pascal addresses many of the synchronization issues that are viewed as problems in other concurrent languages. It is possible to construct complex synchronization schemes for a large class of applications and examples. Synchronization is separated from sequential operation design. Concurrent use of a resource can be encapsulated within an object. Modifications in the number and design of user processes do not affect the implementation of a resource. Path expressions are readable and verifiable.

One of the facilities lacking in Path Pascal is a means to specify scheduling concerns within the language. Scheduling facilities can be constructed using Path Pascal, and Path Pascal programs can be linked to such facilities. However, it is not possible to specify that processes requesting the mutually exclusive execution of a routine embedded within an Open Path Expression should be served in an application-defined order. For example, the application might require that the scheduling order is determined by the number of critical real-time resources being used by each process.

In this section, we discuss scheduling within embedded systems, and consider possible language mechanisms that could be used in Path Pascal. We give a scheduling example based on the implementation of a shortest job first scheduling algorithm and show how this might be specified using scheduling primitives within Path Pascal.

6.1. Embedded Operating System Scheduling Requirements

Any component of an operating system that cannot service all requests as they are received must implement some form of scheduling to choose between pending requests. Scheduling is used to decide which process will gain control of a processor, and to order requests for access to peripheral devices and resources. Scheduling is also an inherent part of any scheme using priorities or deadlines to ensure timely service[Deitel 84].

Many of the performance requirements of an embedded computer system will need to be supported by efficient scheduling. For example, a real-time file system will allow a user to specify a "degree of urgency"[Foudriat et al. 84] for a file when it is opened. This urgency measure must be used within the filesystem *name-server* and *real-time server*[Foudriat et al. 84] to mediate between competing service requests. The loader also must use priority, deadlines and usage information to schedule its work. Scheduling problems also pervade the run-time and communications systems. For this reason, it is very important that scheduling be reliable and efficient.

Very few languages provide scheduling support to users. Users are left to build their own schedulers out of the primitives provided by the language. As T. Wei mentions in his thesis[Wei 81], any concurrent language must implement scheduling to coordinate parallel processes. Scheduling is frequently specified implicitly, as in Path Pascal.

In **Appendix G**, we present a scheduler implemented in Path Pascal. The example will serve to aid discussion of various aspects of scheduling in Path Pascal. A shortest job next scheduler was chosen as the example because it is similar to but simpler than a deadline scheduler and avoids the issues that arise over preemption. The *Shortest Job Next* scheduling algorithm provides the minimum average response time for a group of processes, provided that the running time of each process can be estimated. This can be regarded as a special case of priority scheduling.[Peterson and Silberschatz 83] discuss this algorithm in the context of CPU scheduling.

The design of a non-preemptive shortest job next scheduler for a resource is not intuitively difficult. Each process that calls the resource provides a job time estimate to the scheduler. When the resource is available, the waiting process with the smallest running time estimate should gain access to it. A simple implementation of such a scheduler queues incoming requests in ascending order of job estimates. When the resource becomes free, the least estimate job can simply be dequeued from the front of the list.

The Path Pascal solution presented in **Appendix G** introduces several monitor-like components. The event descriptor object essentially implements a monitor signal/ wait. It functions like a semaphore, and is as error-prone and difficult to use as a semaphore. Consider, for example, how the scheduler would behave if in *resume,* the call to *signal* occurred before the job was dequeued.

Multiple layers of objects contribute to an inefficient solution. Each object uses implicit FIFO queues to implement the Path Expression. On top of these implicit queues, the user must implement an explicit queue to manage job estimate queueing. To implement the user's scheduling requires at least three sets of redundant implicit queues (a set for each of *scheduler, SQ* and *event_dscr*).

Examination of the code for the shortest job next scheduler does not easily reveal which conditions will cause a call to be delayed or resumed. The complexity of the solution appears to exceed the complexity of the problem. A better solution to this problem would:

- encapsulate the resource and the scheduler
- eliminate redundant queues
- reduce the complexity introduced by scheduling

Such examples yield valuable insights into the scheduling requirements for embedded operating systems, and into shortcomings of currently available languages for specifying scheduling.

6.2. Scheduling Support in Higher Level Languages

The problems encountered in designing a shortest job next scheduler are a direct result of the implicit FIFO queueing used in implementing Path Expressions. Scheduling is essentially orthogonal to the synchronization constraints expressed in Path Expressions. As a result, it is possible to design a language mechanism that would allow the user to specify a different scheduling criterion. Research in *scheduling synchronization (see SR[Andrews 81] and[Leinbaugh 81, Leinbaugh 82]) suggests that it is* possible to build efficient user specified scheduling into synchronization.

Such a scheduling mechanism would allow the user to substitute a means of ranking requests for default FIFO queueing. The simplest case would be to choose among competing calls the one with a certain parameter of the least value. A function of parameters and resource state variables should be possible as well. Non-preemptive scheduling can be incorporated into many synchronization schemes. Preemptive scheduling presents possible consistency problems that would need extra support.

Appendix H presents a Path Pascal-like implementation of a shortest job next scheduler that uses such a scheduling mechanism. In the original shortest job next scheduler, the resource was ensured to execute in mutual-exclusion by the use of a specific calling sequence and the semantics of the scheduler. In the new notation, mutual exclusion can be explicitly specified in the resource object.

The appendix shows that scheduling can be applied orthogonally to synchronization without great difficulty. It is also clear that this can fit cleanly into existing synchronization notations. The result of adding a scheduling mechanism results in much cleaner code, and may also have a payoff in efficiency.

A scheduling mechanism of this sort could be applied to many scheduling problems within a real-time operating system, including implementing priorities and deadlines. Where dynamic scheduling criterion were needed, queueing could be done using a function of several variables. The resulting schedulers should not be less efficient or less reliable than the schedulers a user could program[Andrews 81].

7. Distributed Path Pascal

Several enhancements were made to Distributed Path Pascal during the last six months and a new, more reliable, compiler is now under construction. This section describes the improved network facilities now supported by the interpreted Distributed Path Pascal system, and the status of the production Distributed Path Pascal compiler.

7.1. Enhanced Networking

DPP provides *transparent network access* to objects. It is not necessary to specify within a Path Pascal process whether an object is remote or local. Objects are declared as being possibly remote. The compiler turns references to these objects into pointers to objects and uses system calls to dereference the pointers. The object may be local or remote; it makes no difference to the calling process. An object may be changed from one node to another without changing the user programs. This makes the system easier to change, and makes the changes transparent to the applications.

DPP also supports a *transparent network structure.* The underlying nature of the network is hidden from the Path Pascal process. Remote objects are bound through the "import" system call. This system call consults local object tables and queries other processors on the network for instances of the appropriate object. This makes changing the network configuration user and application independent. Changing the network structure will require modifying some system tables, but the user process does not need to know about the change.

With the arrival of the 4.1a BSD and 4.2 BSD systems, a new networking mechanism, *sockets*, was available for use by the DPP interpreter on UNIX. Sockets allow communications between unrelated UNIX processes. This is an improvement over pipes, which require a single process to configure the two ends of the pipe and hand them to offspring processes. In the implementation based on pipes, it is difficult to dynamically add an instance of a Path Pascal Interpreter to a simulation. Sockets do not have this restriction. Once a socket is created, it is bound to a specific address. The socket can then be used to provide datagram service to arbitrary socket addresses (including loop-back service). Each Path Pascal interpreter creates a socket and registers it with a central UNIX process. This UNIX process fields address requests and broadcast messages. In this way, the new DPP interpreter and network support allows Path Pascal Interpreters to be added dynamically to a simulation. When the new interpreter starts up it registers with the central UNIX process; requests for operations on remote objects can proceed normally; and multiple remote objects loaded in the interpreter become accessible to the rest of the network of interpreters.

A single nameserver is not a prerequisite to this scheme. It is possible, and probably advantageous, to build a hierarchy of nameservers (or address-servers). Each nameserver maintains information for a small group of Path Pascal Interpreters. These servers may register with a meta-server which handles communications between the servers in the same way that a server handles communications between the Path Pascal Interpreters.

7.1.1. Networked Path Pascal Interpreters

The initial Interpreter implementation used the UNIX "pipe" interprocess communications. A single UNIX process acted as gateway for all Path Pascal interpreters. Each Path Pascal interpreter sent messages down a pipe to the routing process[2] which passes the message down another pipe (or pipes in the event of a broadcast message) to the recipient(s).

[2] This UNIX process should not be confused with the Path Pascal process concept. A Path Pascal interpreter emulates a Path Pascal machine running many Path Pascal processes within a single UNIX process. A UNIX process can be thought of as a Path Pascal virtual machine.

The Path Pascal interpreter has now been modified to handle remote procedure calls and interface to a network mechanism. The first network implementation was done as part of[Kolstad 83]. Later implementations utilize TCP/IP addressing formats and datagram facilities provided in the 4.2 BSD version of UNIX. The interpreter itself has been modified to isolate the implementation of the particular networking mechanisms. To change networks, approximately 10 routines must be rewritten. The interfacing between these routines and the rest of the interpreter remains unchanged.

Each instance of a Path Pascal interpreter represents a Path Pascal processor. These can run on a single UNIX host or can be distributed across several UNIX machines. The "piped" network was implemented on a single UNIX host; the current implementation allows Path Pascal processes to reside on any number of UNIX hosts.

7.2. A Reliable Path Pascal Compiler

In response to the need for efficient, portable Path Pascal compilers, work has been started on a compiler based on the Pascal system provided with Berkeley UNIX. This compiler generates an intermediate code which is translated by a separate code generator using a machine-independent format. This is the same format that is used by the Portable C Compiler (PCC). Currently there are code generators for the Motorola 68000, VAX-11 family, PDP-11 family, and the National Semiconductor 16000 family of processors. It is believed that this code generator will continue to be ported to new architectures as they are developed, as it provides immediate access to C, FORTRAN and Pascal compilers.

There are several advantages to using the Berkeley Pascal compiler. It uses an LALR(1) parser and has an error recovery ability that the current P4-based compiler lacks. It fixes simple mistakes using a minimum-cost error recovery technique. Additionally, experience with this compiler indicates that it is somewhat easier to modify and maintain than the P4 compiler. Other features of the compiler include separate compilation, the ability to call FORTRAN and C programs directly, and finer control over the selection of hardware data types used to represent Pascal data types.

The current Path Pascal compiler seems to provide a reasonably fast execution environment. In a standard producer-consumer problem run for the production of one thousand items, the process took five seconds of CPU time and eleven seconds of real-time on a fairly loaded VAX-11/750. It must be emphasized that these are timings of the initial, unoptimized implementation which uses only a prototype context-switching mechanism. Since the production/consumption of each item requires two process switches, it would appear that it takes roughly 25ms for an item to be created, several P and V operations to be performed (using a rather expensive method which saves the entire state of the process when performing the semaphore operation and then restores it when done) and the production/consumption of the item to be reported to a UNIX file.

Process switching times vary with the machine, but on the VAX-11/750, it takes about 25 microseconds to perform a context switch. This time is small, and in most applications does not appear to be significant overhead in Path Pascal execution. More important are the schedule, process creation and semaphore manipulation times. One program involving the creation/deletion of a single process 1000 times had a total execution time of 2.8 seconds, or about 2800 microseconds per process creation and destruction. Each creation/deletion cycle involves the execution of four context switches, the use of a generalized dynamic memory package which reclaims process space, and a rather inefficient scheduler.

These execution times can be improved by having the compiler generate machine dependent code for the P and V synchronization primitives. If this approach is taken, non-blocking P operations will be reduced to two instructions on the VAX-11. Similarly, the process creation time is highly dependent on the run-time environment. It is assumed that the final real-time environment will either not use virtual memory or at least provide for memory-locked processes. There are no limits on the number of processes which may be created, which simply depends on the about of memory present. The generalized memory allocation routine may be able to be improved in specialized applications where the memory use pattern is better known.

The new compiler also allows passing procedures, functions, and processes as "formal parameters" to other procedures. This feature is defined in Pascal but was absent from the P4-based implementation. We have also modified the compiler to provide an "otherwise" clause for the "case" statement.

The run-time system, written in the C programming language, provides for dynamic process creation and disposal of process resources (memory and files) upon completion. It is hoped that this run-time system will eventually be re-written in Pascal.

The implementation of the "object" data abstraction facility is nearly complete. Entry procedures, functions and processes are working, as are object initialization procedures and path expressions. The "initially" procedure is also working, and a matching "finally" procedure, invoked when an object is dereferenced, is defined but not yet implemented at the time of this report. Symbol hiding and nesting of objects are all handled in an efficient and logical manner which does not adversely impact the symbolic debuggers. Absolute variable binding is also done.

Possible additions could include the development of global code improvers based on use-definition chaining. This is currently not done in the portable C compiler, but it should be possible to write a code-improver which works on the intermediate code. This has been done by a private company (Zilog), and it may be possible to acquire rights to this program.

7.2.1. Planned Extensions to the Compiler

This re-implementation of the language has provided us with an opportunity to extend the basic Path Pascal system. We are planning on allowing *separate compilation* of objects, provided a skeleton outline is provided. The syntax for remote and "externally declared" or "forward declared" objects will be very similar and will in fact use the same approach internally. The forward declared object concept is being provided to allow cyclic references to objects within objects. Additionally, since more information is kept at compile-time, some improvements in efficiency can be made, such as reducing the space/time product cost of the current "wait-for-sons" primitive.

Other important changes being considered involve implementing deadline scheduling for interrupt processes and modifications to the underlying structure of the distributed object mechanism. It is also hoped that a mechanism can be provided which will allocate the stack-frames for processes from the global heap in a fast manner. This will allow us to remove the need to declare the size of processes and will hopefully reduce the risk of process faults due to lack of stack space.

The networking aspect of the new system is the least developed aspect. Currently, we are investigating using the "Courier" remote-procedure call (RPC) mechanism to provide the RPC's for remote object reference. This would hopefully reduce much of the complexity of the networking software, as well as provide a standardized RPC mechanism which automatically performs machine-dependent data format conversions to provide greater machine independence in a multi-architecture environment. The Courier package was developed by Xerox PARC to provide a reliable RPC mechanism. Courier is layered on top of the NASA/DOD approved Internet Protocol (IP), and thus should enjoy a great deal of support. In an effort to determine if there is a better standard, we are also investigating the standard proposed by Sun Microsystems Inc., based on their experience with the Courier standard.

7.2.2. Debugging Facilities

The most important advantage of using this new compiler system is related to the support environment. The Berkeley Pascal compiler generates sufficient symbol table information and run-time tests to be used with a variety of source-level debuggers (Either "sdb" on 4.1BSD UNIX, or "dbx" on 4.2BSD UNIX). These debuggers provides run-time tracing, triggers on reference to specific variables, line-at-a-time execution, symbolic variable inspection, and break-points on either the line or machine instruction level. Additionally, there is a facility for running profiled programs to determine execution time statistics. The profile facility is an excellent method for determining which sections of programs need to be improved. While none of these debuggers were designed for a multi-process program, we feel that it should be easy to provide the few extensions needed to make debugging such programs simple.

7.2.3. Porting Path Pascal to the IBM Workstations

The portable Path Pascal system is now being ported to the IBM 68000 Xenix system. The compiler has been ported already and is generating P-code. The interpreter will be ported next. The only problems encountered so far can be attributed to incompatibilities between Berkeley Pascal (underlying the VAX Path Pascal implementation) and IBM Pascal.

7.3. Uses of Path Pascal

As a side note, we feel that it should be useful to remark that several research groups on the UIUC campus have been using Path Pascal for simulation of inherently concurrent activities, including advanced processor architecture, and simulation of a fileserver using an Ethernet under a variety of loading conditions.

Prof. Belford's research group is using Path Pascal for simulation studies of different concurrency control protocols for distributed databases.

The research group headed by Prof. Jane Liu and funded by an ARMY DOD Contract, required a simulation model for the performance analysis of algorithms executing in a distributed system. The model developed for this purpose was designed in Path Pascal. The model is designed in three layers: the network layer, the host layer, and the user layer. Each layer is defined by several components which describe completely all data structures, internal operations, protocols and interfaces to other layers. Path Pascal object declarations, process declarations, and path expressions were found to be very useful for designing a flexible yet simple model of a distributed system, and were used as the basic building blocks for defining system components such as physical communication busses, message buffers, hosts processors, job queues, etc. The model was defined in approximately 4000 lines of code, and generated approximately 40K bytes of P-code.

This base of users has provided us with continued feedback on the effectiveness of Path Pascal in simulation, as well as an active user community test-bed for new compilers. This will be used to good advantage in the construction and testing of the new compiler.

7.4. Future Experiments

In conclusion, we feel that this re-implementation of the Path Pascal system provides many advantages over the current implementations. While it is being developed in a UNIX environment, this is not required for actual systems. The current compiler can produce code that will run on a bare machine. Similarly, while the current work is being done for the VAX-11 architecture, the modifications needed to port this to Motorola-68000 systems are already in the compiler for the basic Pascal language, and the machine-specific code in the compiler needed for the Path Pascal extensions are fairly minimal. Most of the new run-time environment is written in C for convenience and does not depend heavily on the host machine. When a Pascal version of this code is written, most of these dependencies will be removed.

As the system continues to mature, tests will be run in a multi-architecture environment using an Ethernet communication link between 68000-based Sun workstations, VAX-11 mainframes, IBM Instruments CS-9000 workstations and possibly Pyramid mainframes. These tests should enable us to spot portability problems and maintain a much higher-quality compiler. Additionally, we hope to construct a "stand-alone" system on a VAX-11/750. This would provide us with a better test-bed for the dead-line interrupt process mechanism as well as provide the ability to test the efficiency of the compiled code for device drivers and other time-critical tasks.

8. Summary

During the period of the project under review, we have studied and designed key aspects required in the development of a family of embedded real-time operating systems. We have considered a framework in which the system can be prototyped, incrementally built, tested, measured, and finally delivered. The framework allows software components to be validated and ported unchanged from the development environment to a production environment. It permits software components to be re-used without change in real-time subsystems, networks of systems, fault-tolerant systems, and protected environments.

The key components of our work include the development of GLOSS, a methodology which simplifies the design of a family of operating systems; INDEED, an environment to support prototyping, incremental development, and dynamic reconfiguration of object-based systems; enhancements to portable interpreted Distributed Path Pascal; and the construction of a reliable Distributed Path Pascal compiler. EOS, the example embedded operating system which we are constructing, will provide system services through object-oriented software components.

We have also examined design and measured the performance of several existing network and distributed operating systems including UNIX United. By porting UNIX United to Berkeley 4.2, we have discovered many design issues and performance concerns that will effect the development of EOS. The experiment has also allowed us to confirm many of our design decisions.

We believe that our current research has produced many new and interesting results. In the next few months, we hope to begin to implement many of our ideas which will enable us to provide further experimental evidence of the validity of our approach.

References

[Ada Reference Manual] *Preliminary ADA Reference Manual.* **SIGPLAN Notices** (June 80) vol. 14, no. 6.

[Anderson & Lee 81] Anderson, T. and P. A. Lee. **Fault Tolerance, Principles and Practice.** Prentice-Hall International, Englewood Cliffs NJ, 1981.

[Andrews 81] Andrews, Gregory R. *Synchronizing Resources.* **ACM Transactions on Programming Languages and Systems** (October 1981) vol. 3, no. 4, pp. 405-430.

[Appelbe & Ravn 84] Appelbe, William F. and A. P. Ravn. *Encapsulation Constructs in Systems Programming Languages.* **ACM Transactions on Programming Languages** (April 1984) vol. 6, no. 2, pp. 129-158.

[Birrell & Nelson 84] Birrell, Andrew and Bruce Nelson. *Implementing Remote Procedure Calls.* **ACM Transactions on Computer Systems** (February 1984) vol. 2, no. 1.

[Brownbridge et al. 82] Brownbridge, D. R., L. F. Marshall and Brian Randell. *The Newcastle Connection or UNIXes of the World Unite!.* **Software - Practice and Experience** (1982) vol. 12, pp. 1147-1162.

[Campbell 83] Campbell, Roy H. *Distributed Path Pascal.* In: **Distributed Computing Systems**, Y. Paker and J. P. Verjus, eds. Academic Press, 1983, pp. 191-224.

[Campbell & Anderson 83] Campbell, Roy H. and T. Anderson. *Practical Fault Tolerant Software for Asynchronous Systems.* **SAFECOMP 83, Third International IFAC Workshop on Achieving Safe Real-time Computer Systems, Pergamon Press, Oxford, England** (1983).

[Campbell & Randell 83] Campbell, Roy H. and Brian Randell. "Error Recovery in Asynchronous Systems", Department of Computer Science Technical Report #1148, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983.

[Cheng et. al. 80] Cheng, W. Y., S. Ray, Robert Bruce Kolstad, J. Luhukay, Roy H. Campbell and Jane W. S. Liu. "ILLINET - A 32 Mbits/sec. Local Area Network", Department of Computer Science Technical Report #1035, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 26.

[Cristian 82] Cristian, F. *Exception Handling and Software Fault Tolerance.* **IEEE Transactions on Computers** (June 1982) vol. C-31, no. 6, pp. 531-540.

[Davis 78] Davis, C. T. *Data Processing Spheres of Control.* **IBM System Journal** (1978) vol. 17, no. 2, pp. 179-198.

[Deitel 84] Deitel, Harvey M. **An Introduction to Operating Systems**. Addison-Wesley, Reading, MA, 1984.

[Eswaran et. al. 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger. *The Notion of Consistency and Predicate Locks in a Database System.* **Communications of the ACM** (November 1976) pp. 624-633.

[Foudriat et al. 84] Foudriat, E. C., W. J. Berman, R. W. Will and W. L. Bynum. "An Operating System for Future Aerospace Vehicle Computer Systems", preliminary), Langley Research Center, NASA, Norfolk, Virginia, 1984, p. 45.

[Horton 79] Horton, Kurt H. "A Fault-Tolerant Deadline Mechanism", M.S. Thesis, Department of Computer Science Technical Report #998, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1979, p. 52.

[Jalote & Campbell 83] Jalote, Pankaj and Roy H. Campbell. "Fault Tolerance Using Communicating Sequential Processes", Department of Computer Science Technical Report #1149, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983, p. 21.

[Joy 84] Joy, William N. *The Scientific Computing Environment: Workstations, UNIX and Supercomputers.* **Colloquium, Coordinated SCience Laboratory, University of Illinois at Urbana-Champaign** (May 1984).

[Kolstad 83] Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems", Phd. Thesis, Department of Computer Science Technical Report #1136, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983.

[LeBlane 84] LeBlane, Thomas J. *Programming language support for Real-Time distributed systems.* **Proceedings, International Conference on Data Engineering** (April 1984) pp. 371-376.

[Leinbaugh 81] Leinbaugh, Dennis W. "High Level Specification and Implementation of Resource Sharing", Technical Report OSU-CISRC-TR-81-3, Ohio State University, Columbus, Ohio, 1981.

[Leinbaugh 82] —. "High Level Description and Synthesis of Resource Schedulers", Submitted to ACM '82: Ohio State University, Columbus, Ohio, 1982.

[Leistman 81] Liestman, Arthur L. "Fault-Tolerant Scheduling and Broadcast Problems", Phd. Thesis, Department of Computer Science Technical Report #1063, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981, p. 98.

[Liskov 83] Liskov, Barbara H. and Robert Scheifler. *Guardians and Actions: Linguistic Support for Robust, Distributed Programs.* **ACM Transactions on Programming Languages and Systems** (July 1983) vol. 5, no. 3, pp. 381-404.

[McKendry & Campbell 80a] McKendry, Martin Steward and Roy H. Campbell. "The Execute Statement: Design, Examples, and Implementation Algorithms", Department of Computer Science Technical Report #1044, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 22.

[McKendry & Campbell 80b] —. "Mechanisms for Protection and Process Control in Operating System Languages", Department of Computer Science Technical Report #1038, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 15.

[McKendry et al. 80] McKendry, Martin Steward, Roy H. Campbell and Robert Bruce Kolstad. "Pathos: A Path Pascal Operating System", Department of Computer Science Technical Report #1016, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 20.

[Mickunas & Jalote 83] Mickunas, M. D. and Pankaj Jalote. "The Delay/Re-Read Protocol for Concurrency Control in Databases", Department of Computer Science Technical Report #1145, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983, p. 13.

[Mickunas et al. 84b] Mickunas, M. D., Pankaj Jalote and Roy H. Campbell. *The Delay/Re-Read protocol for Concurrency Control.* In: **Proceedings, First International Conference on Data Engineering**. IEEE, Los Angles, California, 1984.

[Peterson and Silberschatz 83] Peterson, James L. and Abraham Silberschatz. **Operating System Concepts**. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[Postel 81a] Postel, Jon. "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, USC/Information Sciences Institute, 1981, p. 44.

[Postel 81b] —. "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793, USC/Information Sciences Institute, 1981, p. 85.

[Randell 75] Randell, Brian. *System structure for software fault tolerance.* **IEEE Transactions on Software Engineering** (June 1975) vol. SE-1, no. 2, pp. 220-232.

[Randell et al. 78] Randell, Brian, P. A. Lee and P. C. Treleaven. *Reliability Issues in Computing System Design.* **ACM Computing Surveys** (June 1978) vol. 10, no. 2, pp. 123-165.

[Schmidt 83] Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance", MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983.

[Shrivastva & Panzieri 82] Shrivastva, S. K. and F. Panzieri. *The Design of a Reliable Remote Procedure Call Mechanism.* **IEEE Transactions on Computers** (July 1982) vol. C-31, no. 7.

[Spector 82] Spector, Alfred. *Performing Remote Operations Efficiently on a Local Computer Network.* **Communications of the ACM** (April 1982) vol. 25, no. 4.

[Thompson 78] Thompson, K. *UNIX® Implementation.* **Bell System Technical Journal** (July 1978) vol. 57, no. 6, pp. 1931-1946.

[Wei 81] Wei, Anthony Yu-Wu. "Real-Time Programming with Fault-Tolerance", Phd. Thesis, Department of Computer Science Technical Report #1041, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981, p. 125.

[Wei & Campbell 80] Wei, Anthony Yu-Wu and Roy H. Campbell. "Construction of a Fault-Tolerant Real-Time Software System", Department of Computer Science Technical Report #1042, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 24.

[Zimmermann 80] Zimmermann, H. *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection.* **IEEE Transactions on Communications** (April 1980) vol. 28, pp. 425-432.

APPENDIX A

The Delay/Re-Read Protocol for Concurrency Control in Databases

Presented at International Conference on
Data Engineering

Los Angles
April, 1984

1.INTRODUCTION

The problem of concurrency control in databases has received a good deal of attention in recent years[4,5,8,15,18,21]. Concurrency control is the activity of co-ordinating concurrent access to a database by various transactions, such that the actions of one transaction do not interfere with actions of another. Unrestricted concurrency among database transactions can result in an inconsistent database [5, 9]. Eswaran et. al. [8] proposed a protocol, known as Two Phase Locking, to preserve database consistency.

Two Phase Locking requires that each transaction lock the entity it is going to access. A transaction may request a Read lock or an Update lock on an entity. A lock is "granted" only if no other transaction holds a conflicting lock. Furthermore, each transaction runs through both a "growing phase" and a "shrinking phase". In the growing phase a transaction collects the locks that it requires, and in the shrinking phase it releases them. A transaction cannot request any further locks once it has released any lock. A disadvantage of Two Phase Locking is that deadlock may occur. Deadlock is a major concern in concurrency control [11,23], and usually one or more of the deadlocked transactions must be aborted before processing may proceed. This implies that backup data must be maintained so that if deadlock occurs, transactions may be aborted and "undone", thereby restoring the database to a consistent state.

Many variations on locking protocols have been proposed [2,7], and it has been demonstrated that locking achieves somewhat better results when the database is structured as a hierarchy [12, 18]. The solutions proposed by Thomas [21] and stearns [20] have been found to be special cases of Two Phase Locking [4].

The use of locking to maintain consistency is an entirely *preventive measure* that is, it tries to prevent any view of the database from becoming inconsistent. Two Phase Locking assumes the worst case in which any transaction which *potentially* may conflict with another transaction is synchronized. Since this is a sufficient but not a necessary condition for actual conflicts [3,4] Two Phase Locking tends to be overly restrictive and results in a reduction in concurrency.

Kung [13] proposed a *corrective measure* for concurrency control in an effort to relieve the tight restrictions of locking protocols. In his scheme each transaction works on a private copy of the database and no control is imposed on the actions of any transaction. If, on termination, it is determined that the transaction has operated on a consistent state, the transaction is committed and its changes made permanent. However, if the transaction operated on an inconsistent state, the view of the transaction is "corrected" before its changes are made permanent. The "corrective measure" is to abort the transaction and resubmit it, hoping that it will see a consistent state in the new attempt. In the basic scheme a transaction is prone to repeated abortion. Special measures had to be taken to detect and prevent "starvation" of a transaction.

Conflict Graph Analysis [5,6] is another technique used to increase the degree of concurrency. It also employs a preventive technique, but uses static analysis of the conflict graph to reduce the amount of synchronization needed to ensure that the database remains consistent.

In the present paper we present a new protocol, which employs both preventive and corrective measures. The protocol, which we call the *Delay/Re-Read Protocol,* acts, on the one hand, in a *corrective* fashion by sometimes forcing a transaction to re-read some data; it does so upon recognizing that a transaction has read an inconsistent set of data. The protocol acts, on the other hand, in a *preventive* fashion by sometimes imposing a delay before permitting a transaction to write to the database; it does so upon recognizing that such a write might, at the present time, jeopardize the integrity of the database. A Read request by a transaction is always granted without delay. A Write request may be delayed. The protocol is deadlock-free and no transaction is ever aborted. Consequently, no backup data is needed for the operation of the protocol. The protocol supports a greater degree of concurrency than Two Phase Locking and no transaction is ever delayed indefinitely. This work constitutes a part of a forthcoming thesis [12], and is based upon the preliminary results in [14].

This paper is organized as follows. In Section 2 we present our model of a database system. In Section 3 we define our notion of consistency and present some results relating consistency to the ordering of basic actions of transactions. In Section 4 we present the Delay/Re-Read Protocol and prove that it is both consistent and deadlock-free. In Section 5 we discuss some aspects of the Delay/Re-Read Protocol.

2. SYSTEM MODEL

We consider the database to be a collection of distinct objects with unique identifiers, called *entities*. Assertions, called *integrity constraints* specify the possible values of the entities. Integrity constraints govern the possible interactions of *operations* upon entities. A database which satisfies all of the integrity constraints is said to be in a *consistent* state. A complete specification of the integrity constraints for a database might be very large and it might not have an explicit representation.

In order to formalize our moded, we present some definitions.

We denote the set of entities in the database by "E". Each entity may be read or written *indivisibly.*

Definition 2.1. A *transaction,* denoted T^k, is a set of *actions*

$$T^k = \{t_i^k\}_{i=1}^{p_k}$$

together with a linear ordering* $\leq_{T^k}$, on T^k. $<_{T^k}$ is meant to reflect the temporal ordering of the individual actions of T^k. Each t_i^k is a 4-tuple

$$t_i^k = (k, a_i^k, e_i^k, U_i^k)$$

where

(1) k uniquely identifies the transaction, T^k to which t_i^k belongs

(2) $a_i^k \epsilon \{R, W\}$, called the *operation,* denotes either Read or Write

(3) $e_i^k \epsilon E$ denotes the entity upon which the operation a_i^k is performed

(4) $U_i^k \subseteq 2^E$ (power set of E), called the *Use Set.*

In the case $a_i^k = W$, U_i^k denotes a set of entities which are used to compute the new value of e_i^k. Consequently, we may often use a "function" notation when describing a Write action:

$$t_i^k = (k, W, e_i^k(U_i^k))$$

* Recall: a *partial ordering* $\leq$ on a set X is a subset $\leq \subseteq X \times X$ for which $(a,b) \epsilon \leq$ and $(b,a) \epsilon \leq$ implies $a = b$, and for which $(a,b) \epsilon \leq$ and $(b,c) \epsilon \leq$ implies $(a,c) \epsilon \leq$; $(a,b) \epsilon \leq$ is usually written $a \leq b$; if $a \leq b$ and $a \neq b$ then we write $a < b$; $\leq$ is said to be a *linear ordering* on X if for every $a, b \epsilon X$, either $a \leq b$ or $b \leq a$.

In the case $a_i^k=R$, U_i^k is the empty set. Consequently, we may often omit U_i^k when describing a Read action:

$$t_i^k=(k,R,e_i^k)$$

We require that each transaction be *well formed*, that is

(1) a transaction may read an element at most once;

(2) a transaction may write an element at most once;

(3) all Reads of a transaction must precede all of its Writes;

(4) the *Use Set* for a Write action must include the entity being written, i.e. the new value of an entity depends on its old value (among other things), and;

(5) an entity must be read before it can appear in the Use Set of any write.

Formally, these constraints may be written as follows:

Definition 2.2. A transaction $T^k=\{t_i^k\}_{i=1}^{p_k}$ is said to be *well-formed* if and only if the following conditions hold:

(1) if $t_i^k=(k,R,e_i^k)$ and $t_j^k=(k,R,e_j^k)$ then $e_i^k \neq e_j^k$

(2) if $t_i^k=(k,W,e_i^k,U_i^k)$ and $t_j^k=(k,W,e_j^k,U_j^k)$ then $e_i^k \neq e_j^k$

(3) if $t_i^k=(k,R,e_i^k)$ and $t_j^k=(k,a_j^k,e_j^k,U_j^k)$ then either $t_i^k <_{T^k} t_j^k$ or $a_j^k=R$

(4) if $t_i^k=(k,W,e_i^k,U_i^k)$ then e_i^k is in U_i^k

(5) if $t_i^k=(k,W,e_i^k,U_i^k)$ then for every $y \epsilon U_i^k$ there exists t_j^k for which $t_j^k=(k,R,y)$ and $t_j^k <_{T^k} t_i^k$.

Our model is thus a generalization of Papadimitriou's "two-step restricted" model [17], in which our restrictions (4) and (5) with $U_i^k = \{e_i^k\}$ reduce to Papadimitriou's simpler restriction that an entity must be read before it can be written.

3. CONSISTENCY

We assume that a given transaction, T^k, transforms the database from one consistent state to another consistent state (although the database may temporarily be in an inconsistent state while T^k is executing). Our goal is to allow concurrent transactions, yet ensure that when the transactions complete the database will be in a consistent state.

The notion of concurrent transactions is captured by the following definition.

Definition 3.1. Let $T^1, \ldots, T^n$ be transactions. A *schedule*, S, for $T^1, \ldots, T^n$ is the set of actions

$$S = \bigcup_{i=1}^{n} T^i$$

together with a linear ordering, $\leq_S$, on S, for which for all i, $<_{T^i} \subseteq \leq_S$.

As before, the relation $<_S$ is meant to reflect temporal ordering (with truly simultaneous actions having an "effective" temporal ordering imposed by $<_S$). Since actions of each transaction are performed in the order the transaction requests them, it follows that if $t_i^k <_{T^k} t_j^k$ then we must have $t_i^k <_S t_j^k$; hence the requirement that $<_{T^k} \subseteq <_S$.

For each transaction T^i, we define its *registration*, (i,w), as a request which precedes T^i's Writes and which follows T^i's Reads. As we shall see, the registration for a transaction will actually be an enumeration of its Write Set. We extend $<_{T^i}$ (and correspondingly, $<_S$) to include (i,w) in the obvious way, viz., $(i,R,x)<_{T^i}(i,w)$ and $(i,w)<_{T^i}(i,W,y)$. Moreover, we further extend $<_S$ so that if (i,w) precedes (j,w) in time, then $(i,w)<_S(j,w)$.

The aim of any concurrency control method is to ensure that the schedules performed on the database transform it from one consistent state to another. Serializability [8, 16] has been generally accepted as the consistency criterion for schedules. Serializability holds that a schedule for transactions $T^1, \ldots, T^n$ is consistent if the state of the database after executing the schedule is the same as it would have been had the transactions been executed one after another in some order. Note that the order (corresponding to some permutation $\{\pi_i\}_{i=1}^{n}$ *of* $[1,n]$) is not specified.

Given a schedule S, which satisfies the serializability criteria, we refer to the permuted serial execution $T^{\pi_1}, \ldots, T^{\pi_n}$ as an *Equivalent Serial Schedule (ESS).* Such an ESS is not necessarily unique. A schedule having an ESS will be called a *Consistent Schedule.* Not all the schedules are consistent. A ***concurrency control protocol*** is said to be ***consistent*** if it ensures that the schedule that finally acts on the database (which might be different from the schedule submitted) is consistent.

Since our model requires that each entity be read before it can be written, a schedule S can be checked for serializability using its corresponding ***precedence graph***, G_S [22]. We construct the graph as follows. The nodes correspond to the transactions.

The arcs are determined by the following rule:

If $(i,R,x) <_S (j,W,x)$ or $(i,W,x) <_S (j,W,x)$ or $(i,W,x) <_S (j,R,x)$ for any x, then draw an arc from T^i to T^j.

We note that since $\leq_S$ is a total relation, it follows that the undirected version of G_S is a complete graph.

A schedule S is serializable if its precedence graph is acyclic. It follows that we can find an ESS for S by *topological sorting.*

Clearly the temporal ordering of the registrations induces a serial schedule, $\hat{S}$. If T^i precedes T^j in such a serial schedule, $\hat{S}$, we write $T^i <_{\hat{S}} T^j$. We shall see that the Delay/Re-Read Protocol, using those registrations, produces a schedule, S whose equivalent serial schedule is $\hat{S}$.

Lemma 3.1. Let S be a schedule for well-formed transactions. Then the precedence graph G_S has *no* arc from T^j to T^i if for every (i,W,x) in S, $(j,R,x) \epsilon S$ and $(i,w) <_S (j,w)$ implies $(i,W,x) <_S (j,R,x)$.

Proof. The proof is by contradiction. There are only three ways that G_S can have an arc from T^j to T^i:

(1) $(j,R,x) <_S (i,W,x)$. Since $<_S$ is anti-symmetric, this directly contradicts the hypothesis that $(j,R,x) \epsilon S$ and $(i,w) <_S (j,w)$ and $(i,W,x) <_S (j,R,x)$.

(2) $(j,W,x) <_S (i,W,x)$. Since T^j is well-formed, we have

$$(j,R,x) <_S (j,W,x)$$

As in case 1, the hypothesis yields

$$(i,W,x) <_S (j,W,x)$$

which, by anti-symmetry of G_S, disallows this case.

(3) $(j,W,x) <_S (i,R,x)$. Since T^i is well-formed, we have

$$(i,R,x) <_S (i,w).$$

Also

$$(j,w) \leq_S (j,W,x),$$

so

$$(j,w) <_S (i,w)$$

which, by anti-symmetry of $<_S$, contradicts the hypothesis that $(i,w) <_S (j,w)$.

Theorem 3.2. Let S be a schedule for well-formed transactions. Then the precedence graph G_S is *acyclic* if for every (i,W,x) in S, $(j,R,x) \epsilon S$ and $(i,w) <_S (j,w)$ implies $(i,W,x) <_S (j,R,x)$.

Proof. The proof is by contradiction. Suppose that G_S has a cycle involving nodes $T^{i_1}, \cdots T^{i_k}$ $(k>1)$. Since $<_S$ strictly orders the registrations of the transactions, there is one registration (i,w) among $\{(i_1,w),...,(i_k,w)\}$ which is "earliest" in time. Now for every *other* transaction, T^j, $j \epsilon \{i_1,...,i_k\}$ $(j \neq i)$, we have $(i,w) <_S (j,w)$, which by hypothesis implies $(i,W,x) <_S (j,R,x)$. So Lemma 3.1 applies and there can be no arc to T^i from each such T^j, $j \epsilon \{i_1,...,i_k\}$ $(j \neq i)$. Therefore, the presumed cycle involving T^i is not possible.

Corollary 3.3. Let S be as in Theorem 3.2 and $\hat{S}$ the serial schedule induced by the registrations. Then S is consistent and $\hat{S}$ is an ESS of S.

Proof. Since by Theorem 3.2, G_S is acyclic, it follows that S is serializable and hence consistent. Moreover, any serial schedule having G_S as its precedence graph is an ESS of S. Clearly $\hat{S}$ is such a serial schedule.

Informally, the theorem lays down a sufficient condition to be satisfied by the schedule that will ensure that every transaction sees a consistent state, that is, the set of values returned by the Reads of the transaction is such that it is the same as the set of values of these entities in some consistent database state. This does not imply that all the Reads must be performed on the same consistent state. A Read can be performed on any database state, possibly transitory and inconsistent, but the set of values read by all Reads must be such that all the values *can co-exist* in some consistent database state. Theorem 3.2 specifies the condition when this is satisfied. This theorem is the basis of Delay/Re-Read Protocol.

In the following sections $W_i(x)$ and $R_i(x)$ mean same as (i,W,x) and (i,R,x) respectively.

4. DELAY/RE-READ PROTOCOL

Not all schedules satisfy the condition of Theorem 3.2 in the form they are submitted. The purpose of the Delay/Re-Read Protocol is to control *any schedule* so that the schedule that finally acts on the database satisfies the condition of the theorem.

Each transaction is submitted to a *Transaction Manager* which assigns a *Transaction Process (TP)* to each transaction. A *History File* is used to record the information about the actions performed on the database by the various transactions. This is different from a "log file". A log file, along with data about the actions also records the old and new values of the entities which are modified to provide a "backup". The history file records only a window of activity and no "backup" data is recorded. As we shall see, the history file need only maintain a record of the actions of recent transactions.

When a transaction requests a Read, the TP permits the read and records this action in the history file. No control is exercised over the Read requests. When a transaction requests a Write, the TP executes the protocol and awaits its instruction(s). The protocol may allow the TP to permit the request or may require the TP to re-read some entities, to re-do the computation, and to re-submit the Write request. When the Write is granted the TP permits the Write and records the action in the history file.

The Delay/Re-Read Protocol is used to ensure that any schedule remains consistent. This is accomplished by a combination of preventive and corrective measures. The Delay/Re-Read Protocol sometimes *delays* a Write request (a preventive action). Alternatively, the Delay/Re-Read Protocol sometimes requires TP to re-read some entities prior to proceeding with a Write (a corrective action), thereby assuring that the Use Set for the Write is consistent.

We assume that the Write Set of the transaction is known by the TP. This information is required after the transaction has performed all of its Reads. A similar assumption has been made in SDD1 [6], and is required in locking protocols in order to determine whether to request a shared or exclusive lock. This does not place any restrictions on what may be read and written by transactions, but rather merely requires that a transaction's Write Set be known. After the transaction has performed all its Reads and before it performs its first Write, it records its Write Set in the history file. If the Write Set of T^i is $\{x,y,z\}$, this is recorded in the history file as $\hat{w}_i(x)\hat{w}_i(y)\hat{w}_i(z)$. The recording of the Write Set is assumed to be an atomic action. This action serves the purpose of the *registration* as discussed in section 3.

A Read action is recorded as $R_i(entity\text{-}name)$ in the history file. A Write action of the form $W_i(x(U))$ is recorded as $W_i(x)u_i(x_1)u_i(x_2)$ ... $u_i(x_m)$ where each $x_i \epsilon U$. The writing of this sequence is taken to be atomic. (Note that one of the $x_i{=}x$ and we need not include $u_i(x)$ since it is implied by $W_i(x)$. For the sake of uniformity we will assume that $u_i(x)$ is also recorded)

Let us now present the Delay/Re-Read Protocol formally.

Let $x,y \epsilon E$

The History File, H is maintained as a string over the alphabet

$$\sum=\{R_i(x), W_i(x), \hat{w}_i(x), u_i(x) \mid x \epsilon E\}$$

Let an ellipses (...) denote an arbitrary string over $\sum$ (possibly of length zero).

Let TP(j) be the transaction process of T^j. The Delay/Re-Read Protocol is shown in figure 1.

Given a request for $W_j(x(U))$,

```
(* SECTION I *)
1.  for every $T^i <_{\hat{S}} T^j$ do
2.      { if $H=...R_i(x)...$ & $H\neq...R_i(x)...u_i(x)...$
        then
3.          if there exists $T^k <_{\hat{S}} T^i$ for which $H=...\hat{w}_k(x)...$ & $H\neq...W_k(x)...R_i(x)...$
4.          then await $u_i(x)$
        }

(* SECTION II *)
5.  for every $y \epsilon U$ do
6.      { for every $T^i <_{\hat{S}} T^j$ do
7.          { if $H=...\hat{w}_i(y)...$ & $H\neq...W_i(y)...$
8.          then await $W_i(y)$
            }
        }

(* SECTION III *)
9.  if there exists $y \epsilon U$ & $T^i <_{\hat{S}} T^j$ for which $H=...W_i(y)...$ & $H\neq...W_i(y)...R_j(y)...$
    then
10.     {for every $y \epsilon U$
11.         {if there exists $T^i <_{\hat{S}} T^j$
12.                 for which $H=...W_i(y)...$ & $H\neq...W_i(y)...R_j(y)...$
13.         then instruct TP(j) to reread y
            }
14.     instruct TP(j) to recompute $x(U)$
15.     instruct TP(j) to resubmit $W_j(x(U))$
        }
16. else authorize $W_j(x(U))$.
```

Figure 1: The Delay/Re-Read Protocol

Sections I and II constitute the preventive action of the protocol (causing delays); section III constitutes the corrective action (causing re-reads).

Informally, the Delay/Re-Read Protocol ensures that there is no arc in G_S from T^j to T^i (where $T^i <_{\hat{S}} T^j$). For this the protocol must ensure that for any $x \epsilon E$

(1) $W_i(x) <_S R_j(x)$ (ensured by Section III)

(2) $W_i(x) <_S W_j(x)$ (ensured by Section II)

(3) $R_i(x) <_S W_j(x)$ (ensured by Section I)

Since re-reads are possible, R here means the effective or the final Read. For any $T^i <_{\hat{S}} T^j$, if condition 1) does not hold (line 12 and 7), the protocol ensures that T^j waits until $W_i(x)$ is performed (line 8) and then re-reads the entity (line 13), thus ensuring condition 1).

Condition 2) is satisfied since the well-formedness criterion $R_j(x) <_S W_j(x)$ together with Condition 1) ensure that $W_i(x) <_S W_j(x)$.

It takes a bit more thought to see why it is necessary to do anything more to ensure that Condition 3) is satisfied. A transaction, T^i may not perform $W_i(x)$ if some $T^j <_{\hat{S}} T^i$ will "soon" be instructed to re-read x. This situation is illustrated by the following time line.

$$\begin{array}{llllll}
 & & \longrightarrow \textit{time} \longrightarrow & & & \\
T^k: & R_k(x)\ \hat{w}_k(x) & & W_k(x(x)) & & \\
T^j: & R_j(x)R_j(y)\hat{w}_j(y) & & & \mathbf{R_j(x)}\, W_j(y(x,y)) & \\
T^i: & & R_i(x)\hat{w}_i(x) & [\mathbf{W_i(x)}] \dashrightarrow & & W_i(x)u_i(x)
\end{array}$$

It should be pointed out that in the protocol when we refer to a T^i such that $T^i <_{\hat{S}} T^j$, we can exclude from consideration any transaction T^i which terminated before T^j started, since such a T^i automatically satisfies the conditions of Theorem 3.2. To avoid complication, we do not mention it in the protocol.

We conclude this section with a number of claims for the Delay/Reread Protocol.

Claim 4.1. The Delay/Re-Read Protocol is consistent.

Sketch of Proof. The above discussion illustrates that any $R_j(y)$ occurs *after* all $W_i(y)$ that may occur (for $(i,w)<_S(j,w)$). Thus, the hypotheses of Corollary 3.3 are satisfied, and the resulting schedule is consistent.

Claim 4.2. The Delay/Re-Read Protocol is deadlock-free.

Sketch of Proof. T^j is made to wait for T^i only if $(i,w)<_S(j,w)$. Since $<_S$ is a linear ordering, it follows that no deadlock can occur.

Claim 4.3. For any entity at most one re-read is performed by any transaction.

Sketch of Proof. Before a transaction, T^j discovers in Section III that it must perform a Re-read of some entity y, T^j must first pass through the "gate" of Section II. Section II delays the progress of T^j until all elder transactions, $T^i <_S T^j$ have performed their Writes of entity y. Therefore, upon entry to Section III, transaction T^j is assured that elder transaction have finished their Writes to entity y. Moreover, any *younger* transaction $T^k >_S T^j$ which wants to perform a Write to entity y is delayed in Section I until T^j has performed its Re-read of y.

Claim 4.4. No transaction is delayed indefinitely.

Sketch of Proof. Inspection of the protocol shows clearly that no delay is ever imposed on the eldest transaction. Since we assume that transactions always terminate, it follows that eventually every transaction becomes the eldest of the active transactions, and is therefore immune to further delay.

5. DISCUSSION

History File : It may appear that the history string, H, grows without bound. However, there is a simple method by which we can prune H. We observe that we need not record the actions of any transaction that terminated prior to the start of all currently active transactions. Hence, actions of such transactions can be removed from H. We further observe that the performance of a Re-Read, $R_j(x)$, obviates the need for any previous record of $R_j(x)$; thus H can be further pruned of such $R_j(x)$'s.

History file pruning need not be done by the Protocol or the TPs. A background process can maintain and prune H. Since, the record of actions being removed from H are not being considered by the protocol, the background process will not interfere with the protocol, and so no synchronization is needed for it. This technique will keep the history file pruned and make the act invisible to the protocol while reducing its overhead.

Efficiency Considerations : We may make a few observations concerning the "overhead" of the Delay/Re-Read Protocol. Overhead in the Delay/Re-Read Protocol is of three forms: "delay overhead" corresponding to the delay of a Write in lines 4 and 8;

"re-read overhead" in line 13, which includes the re-computation of $x(U)$ in line 14, and; "search overhead" resulting from the pattern searching in the history file.

It is apparent that if H is pruned, as indicated above, then there is *no* overhead when there is only one active transaction. Moreover, there remains no overhead, even with multiple concurrent transactions, so long as they operate on disjoint sets of entities. Overhead increases only as the interaction among transactions increases, *vis-a-vis* increasingly overlapping Use Sets.

The overhead can be further reduced by utilizing the fact that if an entity has been used in a previous Write by the same transaction, its value is consistent and so the corrective part of the protocol (Section III) need not be executed for that entity.

The search overhead is reduced by pruning H. It can be further reduced by organizing the history file efficiently. For example, since each pattern the protocol looks for is specified by actions on the same entity, we can divide the history file into sub-history files, one for each entity (or a group of entities). Hashing and/or indexing can then be used to further reduce the search time.

The scheme can also be modified to eliminate the recomputation overhead, where the computation is expensive. Suppose T^i attempts $W_i(x(U))$ where the computation of $x(U)$ is expensive. T^i might view $W_i(x(U))$ as merely a request to write, without first performing the expensive computation. Only once the Write has been authorized, does T^i proceed with the computation of $x(U)$, finally performing the Write.

Advantages Over Locking : It is difficult to compare Two Phase Locking with the Delay/Re-Read protocol because both have different overheads resulting from the different strategies followed. Some comparisons are however possible (although we make no attempt here at a full comparison).

1) By using Two Phase Locking transactions can deadlock. The Delay/Re-Read Protocol is deadlock-free.

2) Because of the corrective strategy, the Delay/Re-Read Protocol provides greater concurrency, sometimes at the cost of re-read overhead. But, the Protocol also provides a greater degree of concurrency even without any re-read (see example 1 below). Also, in the Delay/Re-Read Protocol a Write request is delayed *only* when it *would* otherwise lead to inconsistency, while in Two Phase Locking granting a lock can be delayed when it *might* otherwise lead to inconsistency. The throughput of the Delay/Re-Read Protocol is further improved by this fact.

3) Locking requires a lock table, the size of which is fixed and is a function of the total number of entities in the database. This can be a rather large and unnecessary overhead under low concurrency. Moreover, locking also requires a "log file", so that the actions of some transactions can be "undone".

The Delay/Re-Read Protocol needs merely the history file, the size of which depends upon the current degree of concurrency. For low concurrency this overhead is low. Moreover, no backup data need be recorded for the protocol.

4) There is a possibility of 'starvation' in Two Phase Locking, when more than one transaction is waiting to lock an entity. The problem is solved by using so-called fair schedulers. No problem of starvation occurs with the Delay/Re-Read Protocol and no extraordinary measures are needed to prevent starvation.

Two Examples : We give two examples. (left-to-right vertical alignment indicates temporal ordering) First in which no re-read is to be done and no write is delayed. In the second example, a re-read is needed.

Example 1:

T^1:$R_1(x)R_1(y)\hat{w}_1(x)\hat{w}_1(y)$ $W_1(x(x,y))$ $W_1(y(y))$

T^2: $R_2(x)R_2(z)$ $\hat{w}_2(z)$ $W_2(z(x,z))$

Two-Phase Locking would force $R_2(x)$ to wait until after $W_1(x(x,y))$, effectively forcing serial execution, while the Delay/Re-Read Protocol permits full concurrency and requires no delay or re-read overhead. In fact, in this example the Delay/Re-Read Protocol will result in optimum throughput (neglecting the time to execute the protocol).

Example 2:

T^1 : $R_1(y)R_1(x)$ $\hat{w}_1(x)\hat{w}_1(y)W_1(y(y))$ $W_1(x(x))u_1(x)$

T^2 :$R_2(x)$ $R_2(y)\hat{w}_2(x)\hat{w}_2(y)W_2(y(y))$ $\mathbf{R_2(x)}\,W_2(x(x)$

In this example no delay overhead is incurred. But prior to performing $W_2(x(x))$ T^2 must re-read x (shown emboldened), and must recompute $x(x)$ (if it had already been computed using the old value of x). Locking will force serial execution and a simple minded Two Phase Locking protocol will deadlock.

6. CONCLUSION

We have presented a new protocol for controlling concurrent access to a database which uses both preventive and corrective measures for maintaining consistency, and in so doing, permits a high degree of concurrency. The protocol is deadlock-free and accomplishes its "forward recovery" without the need for backup data, without the need for reversing the effects of any Writes, and without aborting transactions. The utility of this method will vary from system to system, depending on the re-read overhead in a particular system. We are currently studying the effects of the underlying system structure on the overhead of the protocol. We are also working on generalizing the protocol for providing different levels of concurrency.

ACKNOWLEDGEMENTS:

The authors would like to thank Prof. Geneva G. Belford for providing useful suggestions. One author (P. Jalote) would especially like to thank her for discussions leading to insights into the problem. This work was supported, in part, by the Highly Available Data Base Project of IBM San Jose Research Laboratory and, in part, by NASA Grant NSG 1471.

REFERENCES

[1] R. Bayer, H. Heller and A. Reiser, "Parallelism and recovery in database systems", *ACM Transactions on Database Systems,* June 1980, pp. 139-156.

[2] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees", *Acta Informatica,* vol 9-1, 1977, pp. 1-21.

[3] P. A. Bernstein and N. Goodman, "Approaches to concurrency control in distributed data base systems", *Proceedings* of the National Computer Conference, 1979, pp. 813-820.

[4] P. A. Bernstein, D. W. Shipman and, W. S. Wong, "Formal aspects of serializability in database concurrency control", *IEEE Transactions on Software Engineering,*

vol. SE-5, No. 3, May 1979, pp. 203-216.

[5] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems" *ACM Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 185-221.

[6] P. A. Bernstein, D. W. Shipman and J. B. Rothnie, "Concurrency control in a system for distributed databases (SDD-1)", *ACM Transactions on Database Systems,* March 1980, pp. 18-51.

[7] C. S. Ellis, "Concurrency search and insertion in 2-3 trees", *Acta Informatica,* vol 14-1, 1980, pp. 63-86.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The notion of consistency and predicate locks in a database system", *Communications* of the ACM Nov 1976, pp. 624-633.

[9] J. N. Gray, R. A. Lorie and G. R. Putzolou, "Granularity of locks in a shared data base", *IBM Research Report,* RJ1654, Sept. 1975.

[10] J. N. Gray, "Notes on database operating systems", in *Operating Systems: An Advanced Course,* Vol 60, *Lecture Notes in Computer Science,* Springer-Verlag, New York, 1978, pp. 393-481.

[11] S. S. Isloor and T. A. Marsland, "The deadlock problem: an overview", *IEEE Computer,* 1980, pp 58-78.

[12] P. Jalote, Ph.D. thesis, University of Illinois, Department of Computer Science, in preparation.

[13] Z. Kedem and A. Silberschatz, "Controlling concurrency using locking protocols" *Proceedings* of the 20th IEEE Symposium on Foundations of Computer Scince, Oct

1979, pp. 274-285.

[14] H. T. Kung and J. T. Robertson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems,* June, 1981, pp. 213-226.

[15] M. D. Mickunas and P. Jalote, "The delay/re-read protocol for concurrency control in databases", *Tech. Rep.,* Dept of Computer Science, University of Illinois at Urbana-Champaign, No. UIUCDCS-R-1145, March 1983.

[16] C. H. Papadimitriou, "The serializability of concurrent database updates", *Journal* of the ACM, Oct 1979, pp. 631-653.

[17] C. H. Papadimitriou and P. C. Kanellakis, "On concurrency control by multiple versions" *Proceedings* of the ACM SIGMOD Conference, 1982, pp. 76-82.

[18] D. R. Ries and M. Stonebraker, "Effects of locking granularity in a database management system", *ACM Transactions on Database Systems,* Vol 2, No. 3, Sept. 1977, pp. 233-246.

[19] A. Silberschatz and Z. M. Kedem, "A family of locking protocols for database systems that are modeled as directed graphs", *IEEE Transactions on Software Engineering,* Nov 1982, pp. 558-862.

[20] R. C. Stearns, P. M. Lewis and D. J. Rosenkrantz, "Concurrency control for database systems", *Proceedings* of the Conference on Foundations of Computer Science, ACM, NY, 1976, pp. 19-32.

[21] R. H. Thomas, "A solution to the update problem for multiple copy databases which uses distributed control", *Proceedings* of COMPCON, 1978, IEEE, NY.

[22] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, 1980.

[23] M.Yannakakis, C. H. Papadimitriou and H. T. Kung, "Locking policies: safety and freedom from deadlock", *Proceedings* of COMSAC79 IEEE, Chicago, pp 286-297.

APPENDIX B

Error Recovery in Asynchronous Systems

Submitted for publication to IEEE
Transactions for Software Engineering

1. INTRODUCTION

The demand for reliable computer systems has led to techniques for the construction of fault-tolerant software systems [6] and [11]. These techniques are intended to ensure that a system fulfills the purpose for which it was constructed despite software faults, hardware faults, and invalid invocations of its functions. Networks of computers, distributed resources, and multiple processors introduce new problems of constructing reliable systems and involve the complex organization and control of error recovery in asynchronous systems [8], [12], [14] and [19]. This paper introduces general principles and a framework for the design of reliable asynchronous systems based on fault tolerance incorporating forward and backward error recovery.

1.1. Fault Tolerance and Error Recovery

A fault-tolerant system is one that is designed to function reliably despite the effects of faults (component or design faults) during normal processing. Such a system detects errors produced by faults and applies error recovery techniques in the form of exceptional mechanisms and abnormal algorithms to continue operation and resume normal computation. However, error propagation may hamper error recovery; the continued operation of a system containing an error can result in the introduction and spread of further errors. Successful fault tolerance must enable the system to continue to function despite error propagation during the time interval, which may be lengthy, between the first manifestation of a fault and the eventual detection of an error.

So called "forward error recovery" aims to remove or isolate specific errors so that normal computation can be resumed [16]. It is accomplished by making selective corrections to a system state containing errors. Because recovery is applied to a system state containing errors, forward error recovery techniques require accurate damage assessment (or estimation) [1] of the likely extent of the errors introduced by the fault.

In contrast, "backward error recovery" aims to restore the system to a state which occurred prior to the manifestation of the fault. Using this earlier state of the computation, the function of the system is then provided by an alternate algorithm until normal computation can be resumed [11]. (In practice, the most recent restorable system state which is free from the effects of the fault may be difficult to determine. In order to find an appropriate system state, a search technique may be used involving iteratively attempting recovery from successively earlier restorable states until recovery is successful.) Because backward error recovery restores a valid prior system state, recovery is possible from errors of largely unknown origin and propagation characteristics. (All that is required is that the errors have not affected the state restoration mechanism.) Backward error recovery may involve considerable time overhead and could require extensive testing of potentially acceptable system states.

Forward and backward error recovery techniques complement one another, forward error recovery allowing efficient handling of expected conditions and backward error recovery providing a general strategy which will cope with faults a designer did not - or chose not to - anticipate. As a special case, a forward error recovery mechanism can support the implementation of backward error recovery [7] by transforming unexpected errors into default error conditions.

1.2. Asynchronous Systems

We assume that all the activities of a computer are composed of the activities of a set of primitive operations ("atomic actions"), each of which has the property of indivisibly advancing the state of a computation. Likewise, we can also consider the activities of systems that are more abstract than a computer (for example, the execution of a system of software components) as being formed from a basic set of primitive atomic actions that have the property of indivisibly advancing the state of that system. An *asynchronous system* is one that is designed so that its activities may consist of two or more independent and simultaneously active primitive atomic actions. Of course, abstract asynchronous systems of software could be executed by a computer that is sequential in operation.

In practice, each primitive atomic action is part of a sequence of actions called a *process* which advances a particular computation (or operation on a system) from an initial state, through a set of successive intermediate states, perhaps to a final state. If atomic actions from different processes may be interleaved or active simultaneously, then the system is often described as having *concurrent processes*. Two or more processes *interact* if they include primitive atomic actions which, reciprocally, modify each other's intermediate states. (Such atomic actions are shared in the sense that they advance the computation of more than one process.)

For fault tolerance to be effective, asynchronous systems require the coordination and synchronization of normal activity with any activity supporting fault tolerance. The errors generated by a fault may propagate from one process to another by interactions or interprocess communication. Moreover, faults may manifest themselves in several processes if the fault is a malfunction of a common element in their respective processors. Control of fault-tolerant mechanisms may be defined by a centralized component of the system or by the system's distributed components. The pattern of interprocess communication may permit one group of processes to recover from a particular fault while other system processes continue to perform their normal activities.

Corresponding to a spectrum of constraints that can be imposed upon interprocess communication, there is a spectrum of error recovery techniques for asynchronous systems. For example, conversations [11] so synchronize a pre-identified group of interacting processes that these processes can perform error detection and error recovery before they communicate to other processes not in the group. The restriction on communication prevents possible error propagation to other processes during the conversation and simplifies state restoration.

Transactions constructed from the interactions of processes using a programmed two-phase commit protocol [9] are co-ordinated so as either to produce a result agreeable to all the constituent processes or to restore all information changed by the transaction to its prior state. Such transactions can have a varying number of constituent processes providing that they all obey the protocol.

If no synchronization is imposed on normal activity, processes may detect errors and attempt to perform error recovery independently of other processes. However, such processes require more complex coordination schemes for fault-tolerant provisions, as in the chase protocol [17] and [25]. Starting from the process that first detects an error, this protocol involves notifying all processes that may have received an error propagated via interprocess communication and/or whose activities are invalidated by the backward error recovery of other processes. Within the restrictions imposed by the protocol, each process may independently and asynchronously proceed with recovery.

The construction of systems with activities that are formed from hierarchies of atomic actions provides a structure for fault tolerance in asynchronous systems [1]. Within the hierarchy, the activities of a group of components are co-ordinated to have the properties of an atomic action using more primitive atomic actions (these properties are described in Section 3.1). For example, the components of a critical section may be co-ordinated to update a set of variables indivisibly by the invocation of appropriate operations on semaphores. There are two reasons why such a hierarchy is a convenient structure. If a fault, resulting error propagation, and subsequent successful error recovery all occur within a single atomic action they will not affect other system activities. Furthermore, if the activity of a system can be decomposed into atomic actions, fault tolerance measures can be constructed for each of the atomic actions independently. Thus, atomic actions provide a framework for encapsulating fault tolerance techniques within modular components.

Although atomic actions have been defined many times in different ways (for example, [8], [14] and [15]) we will use the following definition [1]:

> "The activity of a group of components constitutes an *atomic action* if there are no interactions between that group and the rest of the system for the duration of the activity."

A more formal definition and analysis using occurrence graphs formed from events and relations of causality can be found in [3]. Atomic actions are characterized by the set of events they generate. This set has the property that if any two events within that set are connected by a causal chain of events, all the events in that chain must also reside in the set. An interaction between two activities called A and B would correspond, in the event model, to a causal chain of events between two different events generated by A which passes through at least one event generated by B. For example, a message passed between two activities results in an interaction if an acknowledgment is received from the recipient of the message. (Notice that both definitions are more primitive and less constraining than a definition with the property "all or nothing" [14].)

A *system* has been defined as a set of components which interact under the control of a design [16]. Systems that are designed explicitly so as to synchronize the activities of their components in order to form atomic actions have *planned atomic actions*. The design also determines the way in which the components interact with the environment of the system [1]. The environment of a system is another system which provides input to and receives output from the first system. Such an exchange of information is an *operation*. The activities concerned in an atomic action may be internal to the system or may be operations.

If all the operations on a system involve only planned atomic actions, then that system is an *atomic system* (an exchange of information is not necessarily an atomic action). Such systems may be used as components in the design and construction of other, more complex, systems as if their activities were primitive atomic actions. Systems may also contain *spontaneous atomic actions* that arise fortuitously from the dynamic sequences of events occurring in a system. For the purposes of structuring fault tolerance measures, spontaneous atomic actions are of little value even if they can be easily identified as such.

Planned and spontaneous atomic actions represent the two opposite ends of a spectrum of error recovery techniques and depend upon the extent to which explicit constraints are imposed upon interprocess communication. The conversation is an example of a planned atomic action with which backward error recovery is associated. The chase protocol scheme associates backwards error recovery with a more spontaneous form of atomic action dynamically determined by the protocol from past patterns of interprocess communication and available fault-tolerant provisions. Other error recovery techniques based on atomic actions that are more spontaneous than those of the conversation but less spontaneous than those of the chase protocol exist. For example, the two phase commit protocol explicitly co-ordinates processes entering and leaving a transaction but does not specify which processes are involved.

In the present paper, we introduce principles, structure, and a framework for synchronizing and coordinating forward and backward error recovery in asynchronous systems based on atomic actions. We adopt the definitions of error, fault, and failure introduced by [16] and improved by [1]. A fault-tolerant system includes four constituent activities identified as:

i) error detection;
ii) damage confinement;
iii) error recovery;
iv) fault treatment and continued service.

A fault-tolerant scheme must support all four activities. We first review error recovery in a single process system. Next, we propose a general error recovery scheme for asynchronous systems. Finally, we introduce specific implementation techniques for fault-tolerance in systems.

2. ERROR RECOVERY IN SINGLE PROCESS SYSTEMS

A framework for fault-tolerance can be provided by the notions of exception, exception condition, exception handler, and forward error recovery [1], [13] and [7]. Anderson and Lee provide the diagram in Fig. 1 below to illustrate the framework. A component, pursuing its normal activities, receives a request for a service from another component, performs the service, and returns an appropriate response. The request may be parameterized. The component may service a request by invoking the services of other components.

If a service provided by the component is invoked with an invalid set of parameters, the component may return an abnormal result or *interface exception*. Similarly, if a component fails because it cannot tolerate a fault that it has detected, it may return a *failure exception*. Components that explicitly return abnormal results are said to *signal an exception* to the requesting component. (The exception may have parameters.)

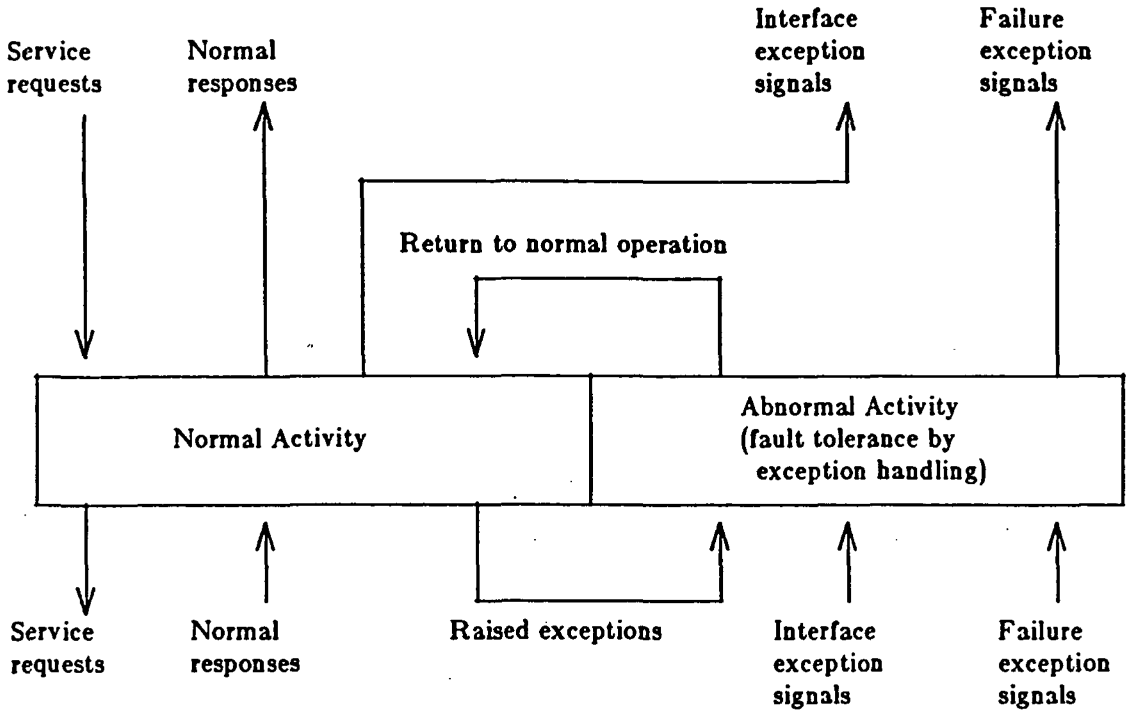

Service requests
Normal responses
Interface exception signals
Failure exception signals
Return to normal operation
Normal Activity
Abnormal Activity (fault tolerance by exception handling)
Service requests
Normal responses
Raised exceptions
Interface exception signals
Failure exception signals

Fig. 1: Framework for an ideal fault-tolerant component.

If a component either receives an abnormal response from an invocation of another component or detects an error or abnormal condition during normal activity, it should *raise an exception* and invoke appropriate fault tolerance measures. Recovery is an abnormal activity of the component and is continued until the component either returns to its normal activities or signals an exception. The relationship between the normal and abnormal activity of a component and the raising and signalling of exceptions is shown in Fig. 1. Note that an exception is raised within the component, but signalled between components.

The flow of control of a computation within a component should change as the result of a raised exception. Such a modified or *exceptional flow of control* is distinguished from the normal flow of control. Within a program, exceptional flow of control is associated with code fragments that are called *exception handlers*. The exception handlers may examine any parameters associated with the exception and provide measures to deal with the exception. Exceptions, software components, and exception handlers are associated by a *handling context*. The *enable* operation creates a handling context and associates it with the current flow of control. The *disable* operation terminates the context. An example of a context nested within another context is shown in Fig. 2. The symbols '[' and ']' represent the enable and disable operations respectively. The notation 'e<x1:h1,...,xi:hi>' and 'd<x1:h1,...,xi:hi>'

represents enable and disable operations with the exceptions x1,...,xi and corresponding handlers h1,...,hi.

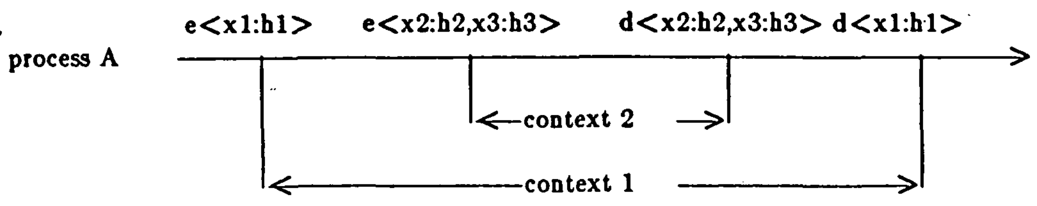


Fig. 2: Example of contexts, exceptions, and handlers.

The measures provided by the exception handler are intended to deal with an exception occurring during the execution of the software component with which it is associated by context. The context may be determined dynamically by the control flow of the program (as in PL/1), by the data flow (as in Id), or statically by scope (as in ADA). Many exception mechanisms use a stack to save contexts. This stack is often coupled with the procedure call mechanism. Careful structuring of the manner in which contexts, components, exceptions, and exception handlers interact can simplify the provision of fault tolerance.

If the fault tolerance measures are successful, a handler may provide a normal control flow return from the component which raised the exception to the component which invoked that component. Figure 3 shows an example of successful forward error recovery in which the relationship between control flow, context, and exception are illustrated.

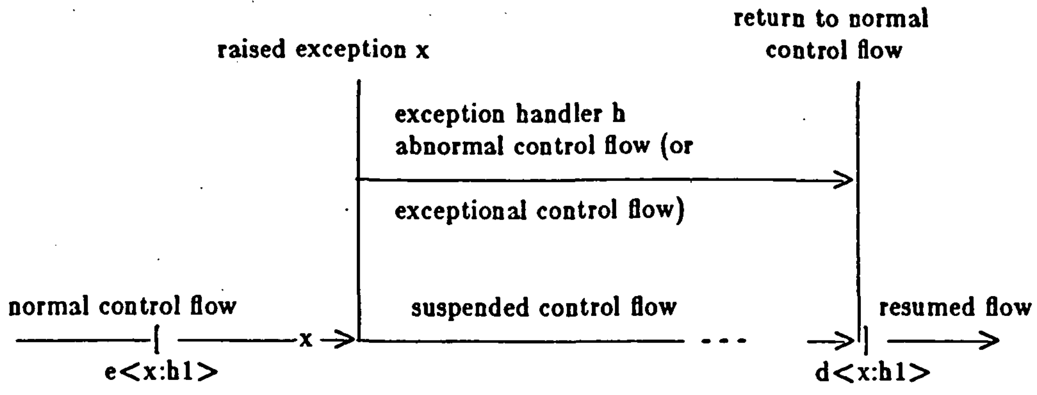


Fig. 3: Example of successful forward error recovery.

If the fault tolerance measures are unsuccessful or inadequate, a handler should signal a failure exception. Abnormal control flow continues in an exception handler of the invoking component. To prevent cyclic and possibly non-terminating patterns of fault and recovery behavior when fault tolerance cannot be achieved, no means is provided whereby a component which receives a signalled exception as a result of an invocation can resume the failed activity [13]. Figure 4 shows an example of returning an abnormal response in which exception handler h2 signals exception failure x1. The component which invoked the failed activity raises exception x1 in response to the signal and invokes handler h1.

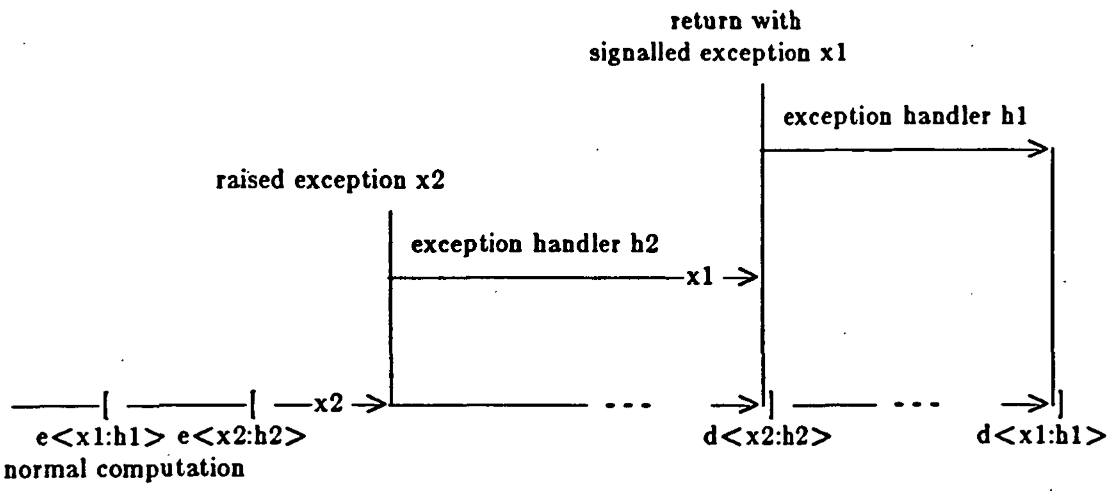


Fig. 4: Example of returning an abnormal response.

An exception handler is a component and may have its own context, exceptions, and exception handlers. This permits the nesting of exception handling facilities.

If an exception is raised within a component (or an exception handler) that does not have a context defining an appropriate handler, the component fails and a failure exception is signalled.

2.1. Exception Mechanisms

Exception mechanisms implement the change in flow of control (or flow of data) implied by the signalling or raising of an exception. Many explicit tests and branches in a software component may be avoided if the exception mechanism is integrated with the interpreter that implements the activity of the component. (For example, the mechanism is often integrated with the operating system or programming language processor). The mechanism may detect a standard set of implicit exceptions (for example, address out of range, divide by zero, invalid operation code) in addition to those raised explicitly by the component.

2.2. Implementing Backward Error Recovery

The framework provided by "exceptions" can be used to implement the recovery block scheme proposed by [11]. (See also [7].) As illustrated in Fig. 5, a recovery block consists of a primary algorithm, one or more alternates, and an acceptance test. On invocation of a recovery block, the primary algorithm is performed and its results are validated by the acceptance test. If, for any reason, the algorithm fails to complete or to satisfy the acceptance test, restoration of a prior system state removes the effects of the algorithm and an alternate is attempted. Each alternate is tried in turn until either a satisfactory evaluation of the acceptance test permits a normal return or the lack of any further alternates requires the signalling of a failure exception.

Figure 6 shows how the recovery block is implemented by a context that includes the primary block, a set of exceptions, and an exception handler. Recursively, the exception handler may have a context, a set of exceptions, and an exception handler. Each recursion implements a particular alternate block. The primary block activates a recovery cache for the preservation of the initial state, executes the primary algorithm, and then applies the acceptance test. If errors are detected, an exception is raised and the exception handler of the primary algorithm is invoked. The exception handler restores the initial state using the cache, attempts an alternate algorithm, and applies the acceptance test. If the acceptance test indicates the presence of errors, the exception handler raises an exception and thus activates its own exception handler. The most deeply nested exception handler signals a failure exception. If any application of the acceptance test indicates a satisfactory result, a normal return is made from the primary or alternate block (exception handler) and hence from the recovery block.

```
ensure acceptance_test
by primary_block
else by alternate_block
else by error;
```

Fig. 5: Example of recovery block.

```
(* ensure correct operation *)
primary_block : system_component
    initialize_cache
    (* start primary *)
    enable(other_exceptions, alternate_1)
    do_primary_algorithm
    if not (acceptance_test) then signal(alternate_exception)
    disable(other_exceptions, alternate_1)
    discard_cache
    return
(* end primary *)


alternate_block_1 : exception_handler
    restore_cache
    enable(other_exceptions, alternate_2)
    do_alternate_algorithm
    if not (acceptance_test) then signal(alternate_exception)
    disable(other_exceptions, alternate_2)
    discard_cache
    return
(* end alternate *)


alternate_block_2 : exception_handler
    restore_cache
    signal(failure_exception)
(* end alternate *)
```

Fig. 6: Equivalent recovery implemented using exception handlers.

This is, in effect, a stylized use of the exception framework to provide backward error recovery. Unexpected exceptions are transformed into the default "other_exceptions" and errors are removed by restoring a prior state using the cache. However, in general, exceptions are used to support forward error recovery schemes which assume detailed knowledge of the erroneous state and attempt to isolate errors. For example, the following forward error recovery scheme is implemented using the exception framework and is taken from recommended take-off emergency procedures for light aircraft.

```
engine_failure : exception_handler
    begin avoid_stall                 (* damage confinement                       *)
    lower_flaps                       (* damage confinement- slower descent       *)
    select_emergency_landing_site     (* damage confinement                       *)
    switch_fuel_tanks                 (* In case of blocked fuel-lines/empty tank *)
    switch_magnetos                   (* In case of ignition system fault         *)
    open_de-icers                     (* In case of fault in iced throttle        *)
    ...
    end
```

Fig. 7: Emergency procedure for light aircraft.

The forward error recovery strategy attempts to land the aircraft safely and thus confine any damage to the engine. Various recovery strategies are attempted to clear possible faults within the aircraft engine.

3. FORWARD ERROR RECOVERY IN ASYNCHRONOUS SYSTEMS

The exception handling described in the previous section of this paper provides a framework for the implementation of fault tolerance in systems with but a single sequential process. Fault tolerance provisions for systems of concurrent processes are complicated by the possibility of communication of erroneous information and the need to co-ordinate processes engaged in recovery. Generalizing the exception handling framework to support fault tolerance in asynchronous systems requires additional system structure concerning the co-operation and co-ordination of the individual processes.

3.1. Structuring Systems of Concurrent Processes

The construction of systems out of components whose activities form atomic actions provides a structure for fault tolerance in asynchronous systems. A system of concurrent processes contains many separate flows of control. Each flow of control represents the sequential activity of one of the processes of the system. Atomic actions, however, involve concurrent activities in which processes communicate in order to co-operate or to co-ordinate their use of shared resources. The flow of control of a process joins or leaves an atomic action at an *entry* or *exit point* of a component, respectively. The system shown in Fig. 8a contains three components each of which is designed so that its activity constitutes a planned atomic action. Figure 8b shows the control flows of two processes that participate in the planned atomic actions. Synchronization is associated with the process entry and exit points of each component in order to ensure that an atomic action occurs.

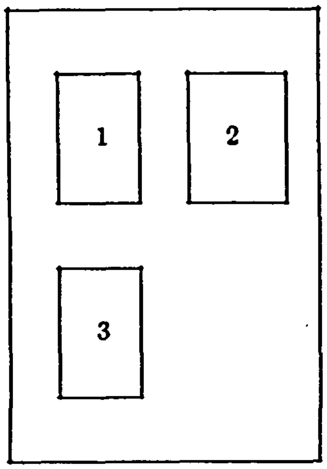


Fig. 8a: Structure of a system with planned atomic action components 1, 2, and 3.

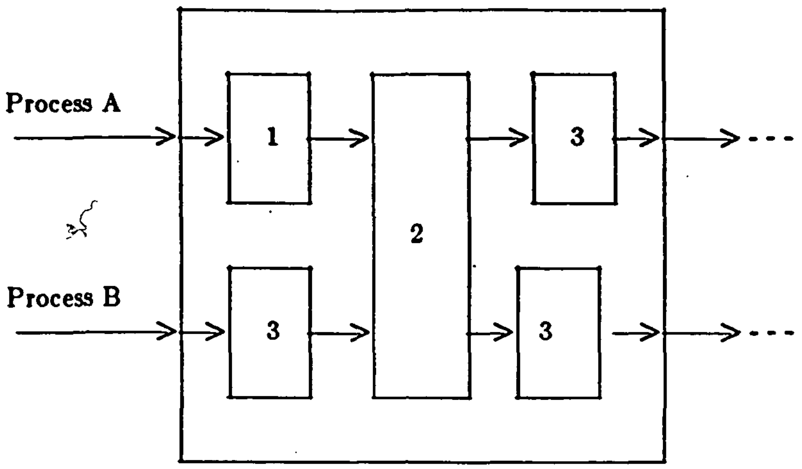


Fig. 8b: The invocation of atomic actions within a system by two processes A and B showing their control flows and entry and exit points.

Fig. 8: An example of a system, its components and processes.

3.2. Fault Tolerance Structuring Principles

If successful, any fault tolerance measures employed within an atomic action are invisible to the rest of the system. This provides a framework for encapsulating such measures into modular components.

The notion of reliability requires that a system have a specification against which the actual results of invoking its operations can be assessed. When an atomic action is executed, a well-defined state exists at the beginning and termination of its activity (although these states may not necessarily be instantaneously observable). The intended relationship between these states constitutes a specification for the atomic action which is independent of any asynchronous activity inside or outside the atomic action.

The reliability of an atomic action depends upon the reliability of each of its components. (Atomic actions which do not contain measures for handling possible faults have been described as being "out of control" [4].) An initial and final state can be associated with the flow of control of each process joining and leaving the atomic action. Pre- and post-conditions associated with such initial and final states can specify the results of the activity of each process. These pre- and post-conditions constitute a decomposition of the specification of the atomic action. The specifications and the encapsulation associated with an atomic action provide a context for the application of error detection and damage assessment techniques. Because atomic actions delimit any error propagation caused by interprocess communication they also support error confinement.

We propose the following two principles for structuring fault tolerance within asynchronous systems:

> 1) The operations provided by a fault-tolerant asynchronous system should be implemented by atomic actions.
>
> 2) Each fault tolerance measure should be associated with a particular atomic action and should involve all of its processes.

A fault-tolerant system is reliable, even though it may suffer from internal faults and contain internal errors, as long as its operations provide services which are in accordance with the system specification. Any fault tolerance measures that the system invokes as a result of detecting such errors should be invisible when that system is used as a component of another system. Hence, system services must be atomic actions. Although this principle appears to restrict the applications for which our techniques are appropriate, in fact this is not the case. Computer hardware and software are often merely components in much larger systems which can be regarded as fault-tolerant, perhaps partly or wholly through the efforts of people and the safety devices of other equipment. Of course, error recovery in such systems must be co-ordinated between components having very different characteristics.

3.3. Exception Handling in Atomic Actions

If a component of an atomic action raises an exception, it indicates the detection of an abnormal condition, or error. The error may have been produced as a result of the activity of this component and/or one (or more) of the other components of the atomic action. Alternatively, the original fault may have occurred prior to the atomic action. The raising of an exception within a fault-tolerant atomic action requires the application of abnormal computation and mechanisms to implement the fault tolerance measures. If the recovery measures succeed, the atomic action should produce the results that are normally expected from its activation. Atomic actions that explicitly return an abnormal result have components that co-operatively signal an exception.

An atomic action may contain internal atomic actions. If an exception is raised within an internal atomic action, then the fault tolerance measures of that internal atomic action should be applied. However, an internal atomic action may signal an exception. This exception is raised in the containing atomic action. An *atomic action failure exception* signifies the failure of one or more of the components of an internal atomic action. In particular, a failure exception should be used to indicate that an internal atomic action did not have an appropriate exception handler for an exception that was raised by one of its components.

ORIGINAL PAGE IS
OF POOR QUALITY

We propose the following exception handling scheme for atomic actions:

Whether one or several components of the atomic action raise an exception, *the fault tolerance measures necessarily involve all of the processes of that atomic action.* (The fact that an exception has been detected elsewhere amongst the processes in an atomic action invalidates the assumptions that any of the processes can terminate normally and provide the appropriate results. If some of the processes are not required to change their flow of control to execute fault tolerance measures, they do not interact with the other processes and hence should participate in a separate atomic action.) Examples of an atomic action in which a component raises an exception and each process of the atomic action changes its flow of control are shown in Fig. 9 and Fig. 10.

Every component of the atomic action responds to the raised exception by changing to an abnormal activity. Each process whose normal control flow is within one of the components changes to an exceptional control flow which executes a handler for that exception. This handler either returns the component to normal activity or signals a further exception. (The change in control flow of a process that occurs as a result of a raised exception in a sequential system is a special case of the changes in control flow that should occur in an asynchronous system.) Figures 9 and 10 show the possible control flows of two processes participating in an atomic actions following an exception. In Fig. 9, the recovery measures implemented by the exception handlers succeed and the normal control flow of the processes is resumed. Figure 10 shows the control flow of the processes of an atomic action when the exception handlers for the components cannot recover.

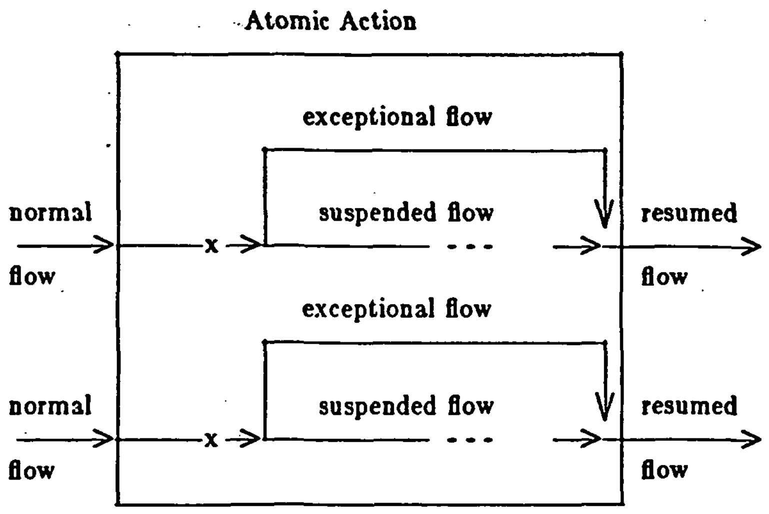


Fig. 9. An example of successful error recovery in an atomic action.

It is convenient to restrict signalled exceptions so that *each component (or exception handler) of an atomic action returns the same exception.* The signalling of the same exception ensures that the components agree on the abnormal result that should be returned to indicate the failure of the atomic action. (Note that an exception should be raised if two or more components try to signal different exceptions. The exception handlers for this exception should signal a failure exception.) An exception is raised in an atomic action if one of its internal atomic actions signals an exception. Signalling a single exception from an internal atomic action simplifies the selection of the appropriate exception handlers and recovery measures.

If *any of the components of an atomic action do not have a handler for a raised exception then all of the components should signal an atomic action failure.*

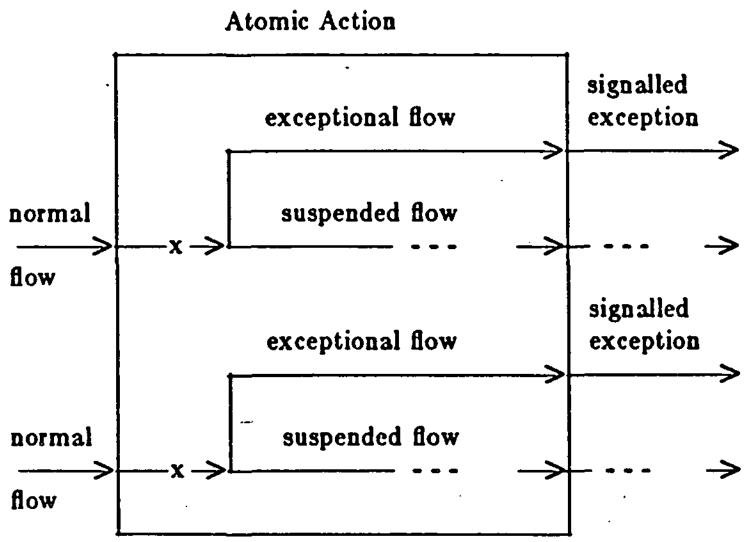


Fig. 10. An example of returning an abnormal response or failure from an atomic action.

The raising of an exception in an atomic action indicates that the computation has reached an erroneous state. The strongest post-condition on the state of an action after an exception is detected is a *damage assessment* predicate. (This definition differs somewhat from [1] because we have chosen to assert what is known about the state of the system containing the errors and faults rather than specify the errors and faults.) The predicate should imply the (preferably weakest) pre-condition for the measures introduced to implement fault tolerance for a given exception. The post-condition for the measures is identical to the post-condition of the atomic action because we have adopted a termination model [13] for the semantics of exception handling.

3.4. An Example from Banking

A bank must ensure adequate cash reserves to allow customers to withdraw money. These reserves are maintained on a day to day basis. If the reserves drop below a certain minimum, the bank will borrow money from other banks until it can redress the balance by either selling assets or gaining new deposits. Depositing or withdrawing money are atomic actions with respect to the day to day management of the cash reserves. If many customers withdraw savings and the bank cannot borrow money for its cash reserves at a reasonable rate of interest, the bank manager may raise an exception in the form of an increase to the bank deposit rate. The error, detected in the form of a lack of cash reserves, could be caused by a number of faults including an increase in the amount that the customers are spending on consumer goods. The customers will now be either earning more money on their deposits or paying more interest on their loans. The customers may invoke exception handlers in the form of transferring more of their money into deposits or seeking a different bank for their loan.

Suppose, however, that some of the customers cannot now afford to pay the interest on their loans. If a large enough amount of money is involved and withdrawals continue, the bank may not be able to cover withdrawals. At this point, the bank must raise a failure exception and suspend customer withdrawal operations. Customers with deposits may now invoke exception handlers in the form of legal action to reclaim their money from the bank.

The bank and its customers interact at two different levels of abstraction. Customers hold accounts with the bank within a fault-tolerant banking system. The banking system provides the service of safely lending money. If reserves drop or the number of customers diminishes, exception measures involving the banking system are invoked and may change the kind of banking service provided (modifying the interest rates or providing additional useful banking facilities like mortgages.) Customers will be aware of such exceptions. They will be particularly concerned if the bank fails. Within this system, the day to day services provided by the bank occur within a fault-tolerant transaction system. These transactions are deposits and withdrawals. Any faults occurring within a transaction (for example, mistakes by the bank teller) should be invisible to the enclosing banking system, since the transactions are intended to form atomic actions.

4. EXCEPTION RESOLUTION

The fault tolerance structuring principles guide the design of a synchronization and co-ordination framework for forward error recovery in asynchronous systems. However, some aspects of the design of the framework are not obvious and need detailed consideration. The concurrent and potentially parallel nature of the execution of the processes within an atomic action may introduce ambiguity in the choice of fault tolerance measures to handle a particular exception condition. Co-ordination between the fault tolerance measures provided by an atomic action and the fault tolerance measures provided by any internal atomic actions also requires careful consideration.

4.1. Resolution of Concurrently Raised Exceptions

Two or more components of an atomic action may concurrently raise different exceptions. This event is likely if the errors resulting from one or more faults cannot be identified with a unique exception by the error detection facilities of each component in the action.

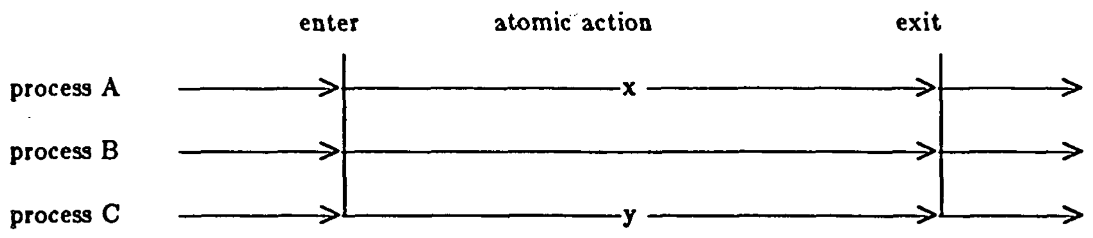


Fig. 11: Exceptions x and y in processes A and C.

Suppose two processes A and C raise exceptions x and y, and that these are different. Two different fault tolerance measures could exist to provide recovery for x and y respectively, each consisting of a set of handlers (that is, a handler for each of A, B, and C). However, the two exceptions, in conjunction, constitute a third exception z: the condition that *both* exception x and y have occurred. A resolution scheme is required to determine the correct recovery strategy. The introduction of an *exception hierarchy* allows resolution of concurrent exceptions within the same atomic action.

Exception Hierarchy

One simple method of providing an exception hierarchy to resolve the ambiguity arising from exceptions that are raised simultaneously is to order the exceptions. From amongst the exceptions raised within the atomic action, the resolution scheme would select the exception with the highest priority in the order. A resolution mechanism would ensure that all the processes in the atomic action change control flow to execute the appropriate handlers associated with this chosen exception. This scheme is frequently used in sequential systems where the state of only one process is involved in error detection within a component but several exceptions may be raised simultaneously. (For example, a power failure interrupt may take precedence over the the detection of an invalid operation code which may take precedence over a page fault for an operand address.) The disadvantage of this simple scheme is that the presence of two or more concurrent exceptions might be symptomatic of a different, more complicated, erroneous state.

We therefore instead propose the use of an *exception tree*. If several exceptions are concurrently raised, the exception used to activate the fault tolerance measures is the exception that is the root of the smallest subtree containing all of the exceptions. This hierarchy permits the occurrence of various exceptions (and thus the detection of erroneous states by several components) to be categorized appropriately by the resolution mechanism.

For example, consider the following exception tree for a twin-engine aircraft:

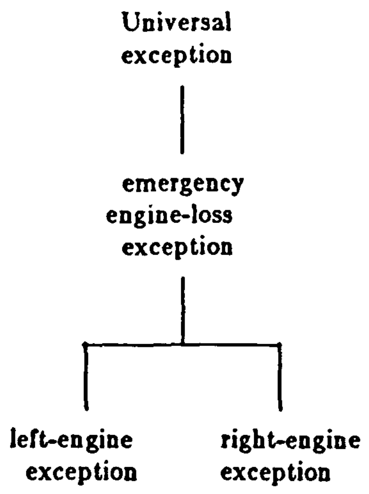


Fig. 12: Example exception tree for twin-engined aircraft.

If the left (or right) engine fails, the pilot can adjust the controls appropriately to compensate for the loss of the left (right) engine in order to fly the aircraft to the nearest airport. If both the right and left engine fail, the pilot must follow the emergency landing procedures. Even with the complete loss of both engines other exceptions could occur that would endanger the emergency landing procedure (for example, fire). All such further exceptions, if not explicitly listed individually within the exception tree, are categorized as the universal exception.

Each atomic action will have its own tree of exceptions. At the root of each tree is, in principle, *the universal exception*. (In practice, the possibility of a 'universal exception' is often ignored.) The universal exception cannot be explicitly raised or signalled by a component; rather it can only be signalled or raised by the underlying exception mechanism. In general, the damage assessment for an exception in the tree will imply the intersection of the damage assessments for the exceptions in each of its subtrees. The greater the number of different exceptions raised within an atomic action, the less the damage assessment predicate may assert about the current state of the system. The damage assessment for the universal exception must assume that any and perhaps all of the state variables and even the representation of the process may have been been corrupted. Only the underlying exception mechanism itself should be presumed undamaged. In contrast, the leaves of the exception tree may have very detailed damage assessments corresponding to the failure of particular internal atomic actions or the detection of specific abnormal conditions and errors.

The exception handlers invoked as a result of several exceptions raised concurrently should have weak enough pre-conditions (that is, equal or weaker than the damage assessment) to allow them to provide the appropriate fault tolerance measures. If the universal exception is raised, its handler should signal a failure exception. The exception mechanism raises the universal exception for any exception that is raised which does not have a handler. However, if there is no handler for the universal exception, the exception mechanism must act on behalf of the atomic action and signal the universal exception to avoid circularity. Even backward error recovery measures require a stronger pre-condition than that provided by the universal exception. The damage assessment predicate for the "other_exceptions" exception (introduced in section 2.3 to help specify backward error recovery using the exception framework) assumes that the cache correctly holds the prior state of the system. In general, exceptions that invoke backward error recovery measures will be descendents of the universal exception and ancestors of any exceptions invoking forward error recovery measures. Some of the leaves of the exception tree may be failure exceptions of internal atomic actions.

It is convenient to provide *default* exceptions and handlers for specific exceptions or classes of exceptions that may occur within the atomic actions of a system. The exception mechanism must ensure that the correct default handlers are enabled during the activation of each atomic action. Examples of default exception handlers are backward error recovery for the 'other_exceptions" exception, forward error recovery that signals a distinguished failure exception for the universal exception and diagnostics for the class of failure exceptions.

The example of nested atomic actions shown in Fig. 13 below implies the set of exception trees shown in Fig. 14.

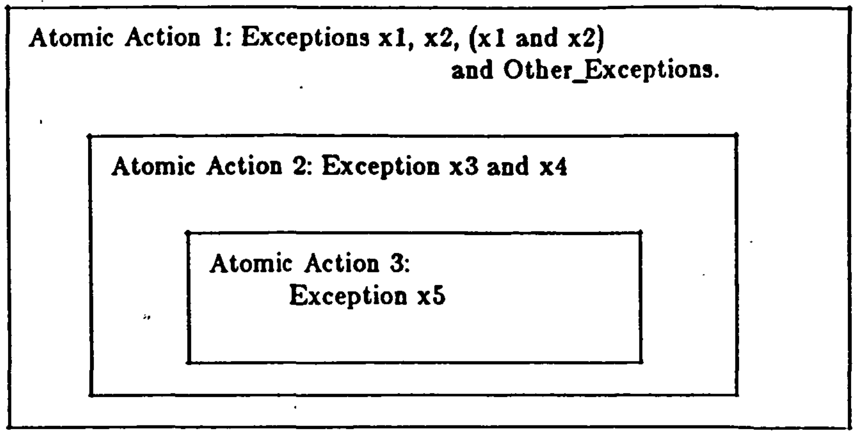
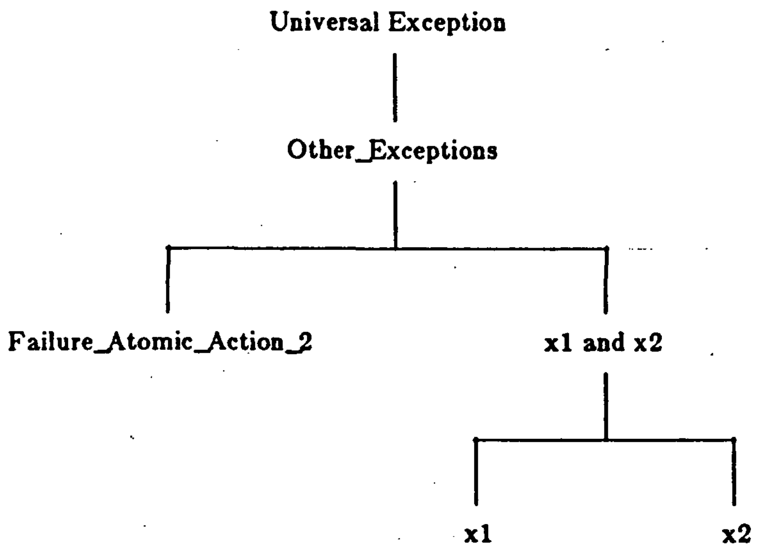

Atomic Action 1: Exceptions x1, x2, (x1 and x2) and Other_Exceptions.
Atomic Action 2: Exception x3 and x4
Atomic Action 3: Exception x5

Fig. 13: Nested contexts of three atomic actions.

Atomic Action 1:



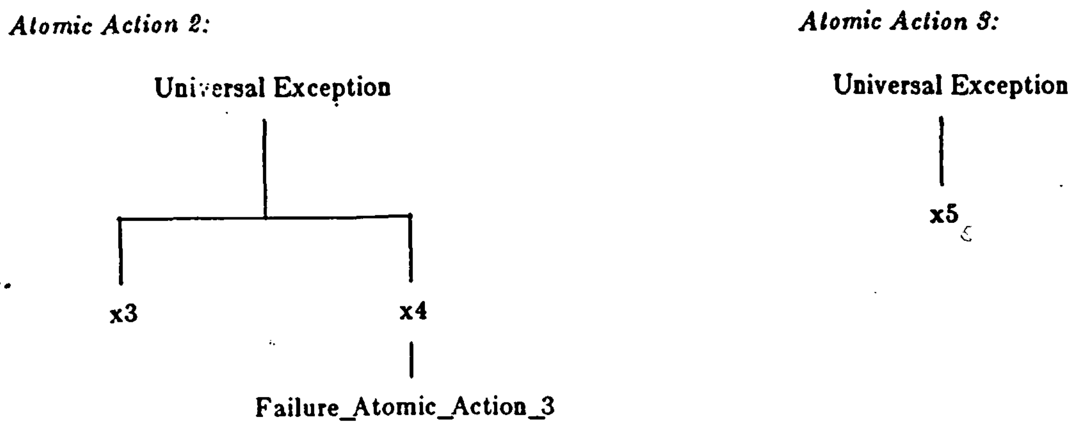


Fig. 14: The three exception trees of the atomic actions.

The default 'Other_Exceptions" exception might be associated with backward recovery measures in the form of a conversation. Exceptions x1, x2, x1 and x2, Failure_Atomic_Action_2, x3, x4, Failure_Atomic_Action_3, and x5 might be associated with specific forward error recovery measures.

The exception tree could be generalized to a complete lattice [18]. The lattice would represent a partial ordering of the exceptions. The resolution mechanism would resolve concurrently raised exceptions by selecting the exception that is their least upper bound within the lattice. The least upper bound of all the exceptions in the lattice would be, of course, the universal exception. Whether such a general structure is desirable for constructing reliable systems can only be determined from future practical experience.

4.2. Exceptions and Internal Atomic Actions

The forward error recovery framework for asynchronous systems must synchronize and co-ordinate recovery from exceptions within fault-tolerant systems that are themselves constructed from fault-tolerant components. In particular:

1 A component of the atomic action may raise an exception while other components of the atomic action are involved in internal atomic actions.

2 The fault tolerance measures for an atomic action may require that internal atomic actions be aborted.

4.2.1. Exceptions and Internal Atomic Actions.

A component of an atomic action raises an exception while other components are involved in internal atomic actions. In principle, all the components of the atomic action must invoke fault tolerance measures even if the atomic action includes internal atomic actions. However, the definition of atomicity makes internal atomic actions indivisible.

In addition, out of the large number of possible exceptions that might be raised within an atomic action, many will have no meaning within an internal atomic action.

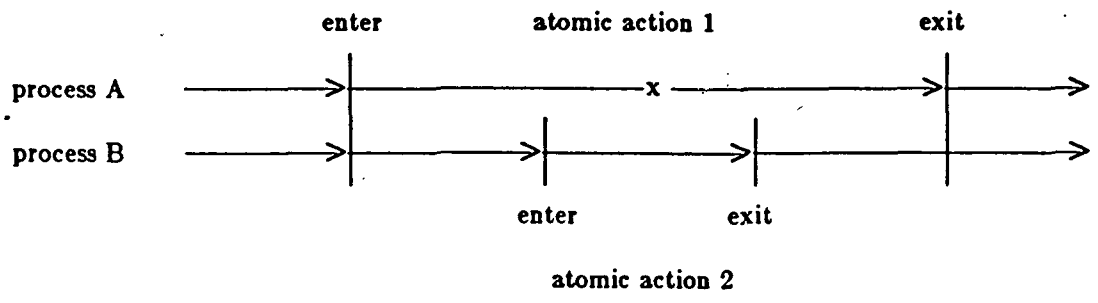


Fig. 15: Example of an exception x in an atomic action with an internal atomic action.

The fault tolerance measures implemented for exception x in atomic action 1 will assume that either the atomic action 2 has not yet started or that it has already completed. Further, exception x may have no meaning within atomic action 2.

Thus, after the detection of an exception, any active internal atomic action must be completed before the fault tolerance measures of the containing atomic action are invoked. (This also implies that if components of an atomic action and components of a internal atomic action concurrently raise different exceptions, the fault tolerance measures of the internal atomic action will be completed first.) However, in certain circumstances it may be desirable to abort an internal atomic action and this situation is examined next.

4.2.2. Aborting Internal Atomic Actions

Although, in theory, a containing atomic action can always compensate for interior atomic actions by masking their effects, these fault tolerance measures cannot be commenced until the internal atomic actions terminate. Thus, if an exception associated with real-time concerns is raised in the containing atomic action, the delay caused because internal atomic actions are active may prevent a timely recovery. Alternatively, the containing atomic action may detect an exception which indicates that its internal atomic actions will not terminate (for example, a deadlock condition). In this case, it would never be able to invoke its recovery measures. The fault tolerance framework must therefore permit the abortion of internal atomic actions. We thus propose the following solution.

An internal atomic action may be aborted if it is defined to have a distinguished abortion exception. An *abortion exception is raised by the exception mechanism to indicate that an exception has been raised in the containing atomic action.* The abortion exception is, to all intents and purposes, a special interface exception that is raised automatically to indicate that the pre-conditions under which the internal atomic action was invoked are invalid. If an abortion exception is raised, the internal atomic action should proceed to apply fault tolerance measures to abort itself. When the internal atomic action has completed its fault tolerance measures and terminates, the containing atomic action may invoke its own fault tolerance measures.

If an internal atomic action cannot abort itself correctly and return normally, it may signal an exception. (If the atomic action can neither return normally nor signal an exception it is not fault-tolerant.) The proposed scheme does not distinguish between a signalled exception returned from an aborted atomic action and one returned from the completion of an ordinary internal atomic action. In both cases, the signalled exception is raised in the containing atomic action and may influence the choice of the fault tolerance measures that are subsequently invoked.

The principle of recovery occurring within an atomic action is not contravened by the abortion scheme because:

1 the abortion exception is included as part of the specification of the internal atomic action;

2 any recovery instituted as part of the abortion of the internal atomic action is accomplished within that atomic action;

3 any recovery instituted by the internal atomic action is indivisible with respect to the containing environment. (The processes in the containing environment are suspended, if necessary, until the internal atomic action completes its recovery.)

4 only one abortion exception is allowed for each internal atomic action and it is raised if any exception occurs in the containing atomic action. *Complete* damage assessment of an atomic action can only be made when all of its processes suspend their normal control flow (and all internal atomic actions have completed). If multiple abortion exceptions were permitted and were bound to different exception conditions in the containing atomic action then:

4.1 an aborted internal atomic action could signal an exception which aborts another internal atomic action. (This complicates the exception mechanism and may impose undesirable delays upon error recovery while internal atomic actions abort each other.)

4.2 an abortion exception could be raised in an internal atomic action that is already trying to abort itself. (That is, the initial abortion of an internal atomic action is based on an *incomplete* damage assessment.)

The abortion scheme we propose requires an internal atomic action to be defined with an explicit abortion exception. However, sometimes it may be more convenient to make the abortion exception implicit and to associate a default handler with the exception. For example, backward error recovery schemes often assume a default abortion exception and handler. If an exception is raised within a conversation, provided a recovery cache permits restoration of prior system states, any internal conversations may be immediately aborted.

4.3. Implementing Backward Error Recovery

The framework we propose permits the implementation of the conversation scheme proposed by [20]. Each conversation is an atomic action composed of several components which have contexts, exceptions and exception handlers. The fault tolerance measures require the activation of a cache for the initial state of the atomic action. The acceptance test provides error detection and raises an exception if it fails. Should an exception be raised in one or more components, normal control flow is suspended in all the processes and exceptional control flow is commenced. The handlers restore changed cached values and attempt the alternate algorithms. If all the alternates fail, a failure exception is signalled to a containing atomic action. We assume that the processes invoking the conversation are appropriately synchronized to execute the conversation correctly, although we do not show the mechanisms concerned.

Example of a Conversation

The interaction of two processes A and B in a conversation is shown in the diagram that follows. Individual actions of each process and combined actions of both processes are distinguished by enclosing them in boxes. Synchronization and co-ordination between the processes is implied by the structuring of the diagram.

enter_conversation_of_A_and_B ensure acceptance_test by	
primary_of_A	primary_of_B
else by	
alternate_1_of_A	alternate_1_of_B
else by	
error	error
exit_conversation_of_A_and_B	

Fig. 16: A conversation.

primary: of_A	primary: of_B
initialize_cache_for_A_and_B enable(other_exceptions,alternate_1)	
do_primary_A	do_primary_B

if not (acceptance_test) then
signal(alternate_exception)
disable(other_exceptions,alternate_1)
discard_cache	

return_A end primary	return_B end primary

Fig. 17: Primary implementations.

alternate_1:of_A	alternate_1:of_B
restore_cache_for_A_and_B enable(other_exceptions,alternate_2)	
do_alternate_1_of_A	do_alternate_1_of_B

if not (acceptance_test) then
signal(alternate_exception_2)
disable(other_exceptions_alternate_2)
discard_cache	

return_A end alternate_1	return_B end alternate_1

Fig. 18: Alternate implementations.

alternate_2: of_A	alternate_2: of_B
restore_cache_for_A_and_B signal(failure_exception)	
end alternate_2	end alternate_2

Fig. 19: Default alternate implementations.

It is trivial to generalize this implementation of the conversation so that internal conversations can be aborted by an abortion exception.

4.4. Recovery Schemes Supported by the Framework.

The exception framework supports both backward and forward error recovery schemes. Within a particular atomic action, it is possible to employ both schemes. For example, particular exceptions may have forward error recovery handlers and any other exception may have a default handler that implements a backward error recovery scheme. Thus, the framework generalizes to the case of asynchronous systems the scheme of combining forward and backward error recovery for a sequential process described in [16].

5. A SPECTRUM OF ERROR RECOVERY TECHNIQUES

Depending upon the extent to which atomic actions are planned or spontaneous, the framework of fault tolerance based on atomic actions supports a spectrum of error recovery techniques (described in Section 1.2) for asynchronous systems.

The *resolution mechanism* automates the propagation of an exception occurring in one process to all other processes in the atomic action while resolving any ambiguities caused by several components raising different exceptions concurrently. It separates the provision of the underlying resolution and exception facilities from the user and simplifies the construction of recovery measures from the exception handling framework. The method by which the processes of an atomic action may be identified depends upon the way in which atomic actions are implemented and the degree to which their constituent activities are parameterized in the definition of the atomic action. Essentially, the mechanism:

1 Suspends all the processes engaged in the atomic action.

2 Aborts any internal atomic actions requiring abortion because of the exception condition.

3 Chooses the appropriate exception that reflects the erroneous state of the atomic action.

4 Raises the chosen exception in all of the processes, causing them to change control flow and execute the appropriate exception handlers.

Note that atomicity prevents the suspension of a process engaged in an internal atomic action until that action terminates. Also, exceptions may be raised concurrently during the interval between the occurrence of the first raised exception and the completion of the last internal atomic action.

We will examine an implementation of the resolution mechanism for two different forms of planned atomic action. The first form of planned atomic action assumes that the identity of the atomic action is explicitly defined (for example, as in a conversation).

The second form of planned atomic action, which provides more spontaneity, allows atomic actions to be implicitly created during the lifetime of a system (for example, as in the chase protocol scheme).

5.1. Explicitly Defined Atomic Actions

Explicitly defined planned atomic actions have pre-determined components that are designed to synchronize to form a particular atomic action. Examples include (i) the checkpointing of a large distributed data base, (ii) the organization of compiled code for a multi-processor, and (iii) computer architectures with uninterruptable instructions.

The synchronization required to implement a planned atomic action may impose an overhead on the efficiency with which internal actions are executed, or may restrict the degree of parallelism between those actions. For example, if concurrent access to a set of resources is infrequent, any synchronization imposed to achieve atomicity is largely a performance overhead. Similarly, mutually exclusive access to a set of resources provides atomicity but eliminates parallel processing.

However, planned atomic actions simplify certain implementation concerns when they are used as a framework for fault tolerance. Prior knowledge of the components of a planned atomic action aids:

1 identification of the components within the atomic action.

2 communication required to co-ordinate invocation of fault tolerance measures.

3 rigorous specification of the function implemented by the atomic action. Rigorous specifications support error detection and damage assessment.

A range of synchronization and co-ordination techniques may be used to construct planned atomic actions. One component of the action may provide a centralized control for synchronizing the other components. Such schemes are similar to the monitor [10] and to similar synchronization schemes for accessing shared data. Alternatively, each component may support a distributed control scheme perhaps based upon transmitting messages and employing a two-phase protocol [9]. Such a mechanism is outlined in the Appendix.

5.2. Implicitly Defined Atomic Actions

Atomic actions which are implicitly defined are less easy to use to support forward error recovery although they have been proposed to support backward error recovery in the chase protocol scheme [17]. The difficulty in using such atomic actions arises from the problems of specifying their correct behavior for the purposes of measuring reliability. Despite the problems of using spontaneous atomic actions in practice, we shall briefly examine the consequences of spontaneity upon the proposed exception handling scheme and resolution mechanism.

The group of components constituting a implicit planned atomic action is recognized *by the interactions that have occurred* and the handling contexts established by the components. The boundaries of the atomic action will coincide with the boundaries of the individual components and contexts that provide fault tolerance measures. Implicit atomic actions introduce a new ambiguity in the choice of fault tolerance measures to handle a particular exception condition. The ambiguity arises because there may be no pre-determined relation between a particular exception and a given atomic action. Consider the following interaction:

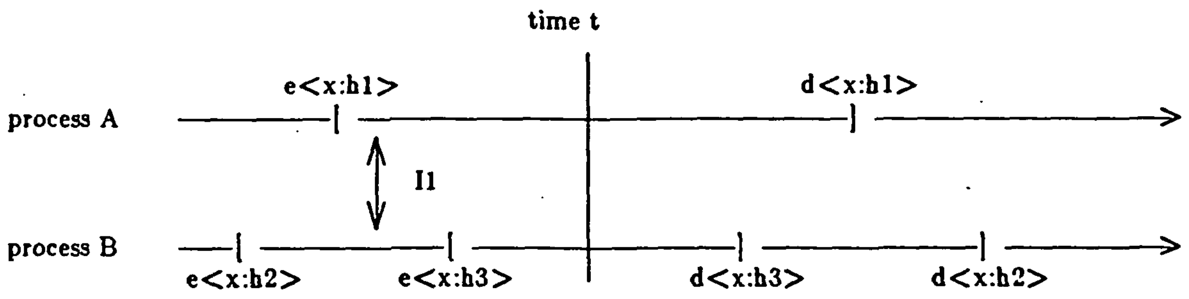


Fig. 20: Exception in implicit atomic action.

The contexts for the handlers h1, h2, and h3 define the boundaries of possible implicit atomic actions that may be used in recovery schemes. Process A and B exchange information at interaction I1 and, by definition, at this point they must be in the same atomic action. Both processes have handlers for an exception x and the enable and disable operations for this exception now delimit an implicit atomic action. If process A raises an exception x at time t, the appropriate context associates handler h1 with the recovery to be invoked. Although exception x occurs when process B is executing within the context associated with handler h3, there is no interaction of that context with process A and it can be regarded as a different implicit atomic action. Therefore, that internal atomic action should be completed and then handler h2 of the enclosing context of process B invoked for the exception x. (If the exception x had been detected in process B instead of process A, only handler h3 would be invoked and process A would be unaffected.)

Any resolution mechanism for implicit atomic actions must compute a recovery strategy from the known set of interactions and the contexts in which they occurred. Such a resolution mechanism will be similar, in many ways, to the mechanism underlying the chase protocol [25].

Commitment Exceptions and Failures

A direct result of the implicit formation of atomic actions is that a process may discard provisions for forward error recovery prematurely. For example, it may disable a handler for a context containing an interaction with another process even though that other process might still raise an error as a result of the exception. Figure 21 illustrates such a situation.

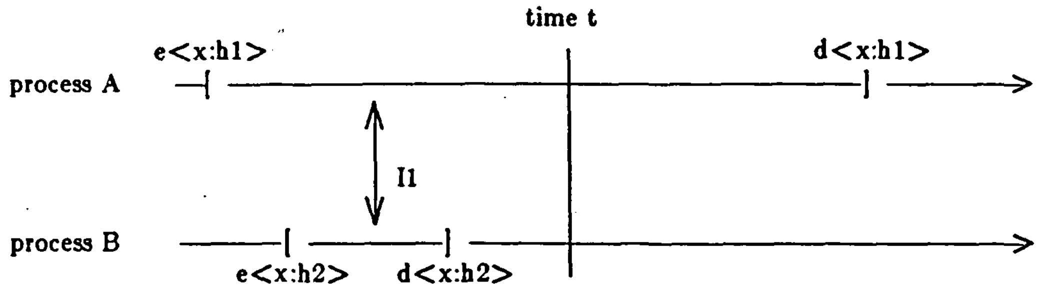


Fig. 21: Example of a commitment exception.

Exception x raised in process A will generate a failure exception in process B which has discarded its exception handler (h2) for x. (This assumes B does not have a handler for x in a containing context.) Such exceptions result from process B committing itself too early to the results of a computation [22] that were formed in a co-operative atomic action. Because implicit atomic actions are involved, it is very difficult to devise a practical forward recovery strategy for commitment errors.

6. CONCLUSION

We have introduced a framework for the provision of fault tolerance in asynchronous systems. The proposal generalizes the form of simple recovery facilities supported by nested atomic actions in which the exception mechanisms only permit backward error recovery, as has been proposed for data bases [4]. It allows the construction of systems employing both forward and backward error recovery and thus allows the exploitation of the complementary benefits of the two schemes. Backward recovery, forward recovery, and normal processing activities can occur concurrently within the organization proposed.

We believe that a reduction in the complexity of the design of fault tolerant software for an asynchronous system can be achieved by using atomic actions to structure the activity of the system. Although many notations have been devised for error recovery which include explicit definitions of atomic actions [2], [12], [14], [15], [21], [22] and [24], most of the notations are either inadequate or too restricted to permit their use as the basis for the exception scheme we have described. Practical systems can only be constructed if suitable notations are developed to express the concept of an atomic action [5].

We have generalized exception handling to provide a uniform basis for fault tolerance schemes within the atomic action structure. The generalization included a resolution scheme for concurrently raised exceptions based on an exception tree and an abortion scheme to permit the termination of internal atomic actions.

Finally, we have outlined an automatic resolution mechanism for exceptions in atomic actions which allows users to separate their recovery schemes from the details of the underlying algorithms. While we have not discussed implementation in any detail in this paper, the mechanism can be implemented with distributed control by means of a simple message passing system.

Acknowledgments. This research was funded by the Science and Engineering Research Council of the United Kingdom who supported Roy Campbell with a Senior Visiting Fellowship while at the University of Newcastle upon Tyne for the 1982/83 academic year.

7. REFERENCES

[1] T. Anderson and P.A. Lee, *Fault Tolerance, Principles and Practice,* Prentice-Hall International, Englewood Cliffs NJ, 1981.

[2] T. Anderson and M. R. Moulding, *"Dialogues for Recovery Coordination in Concurrent Systems,"* Technical Report, In preparation, Computing Laboratory, University of Newcastle upon Tyne, 1983.

[3] E. Best and B. Randell, *"A Formal Model of Atomicity in Asynchronous Systems,"* Technical Report 130, Computing Laboratory, University of Newcastle Upon Tyne, December 1980.

[4] L. A. Bjork and C. T. Davies, *"The Semantics of the Preservation and Recovery of Integrity in a Data System,"* IBM Technical Report TR02.540, December 1972.

[5] R. H. Campbell, T. Anderson and B. Randell, "Practical Fault Tolerant Software for Asynchronous Systems," *SAFECOM 83,* Cambridge, To be published, 1983.

[6] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing,* Toulouse, pp. 3-9, June 1978.

[7] F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers, Vol. C-31, No. 6,* pp. 531-540, June 1982.

[8] C. T. Davies, "Data Processing Spheres of Control," *IBM Systems Journal, Vol. 17, No. 2,* pp. 179-198, 1978.

[9] J. N. Gray, "Notes on Data Base Operating Systems," In R. Bayer, R. M. Graham and G. Seegmuller (Ed.), *Lecture Notes in Computer Science, Vol. 60,* Springer-Verlag, Berlin. pp.393-481, 1978.

[10] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM, Vol. 17, No. 10,* pp.549-557, October 1974.

[11] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery," In E. Gelenbe and C. Kaiser (Ed.), *Lecture Notes in Computer Science, Vol. 16,* Springer-Verlag, Berlin. pp.171-187, 1974.

[12] K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering, Vol. SE-8, No. 3,* pp. 189-197, 1982.

[13] B. H. Liskov and A. Snyder, "Exception Handling in CLU," *IEEE Transactions on Software Engineering SE-5(6),* pp.546-558, November 1979.

[14] B. Liskov, "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering, Vol. SE-8, No. 3,* pp. 203-210, May 1982.

[15] D. B. Lomet, "Process Synchronization, Communication and Recovery Using Atomic Actions," *SIGPLAN Notices, Vol. 12, No. 3,* pp. 128-137, March 1977.

[16] P. M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling," *SIGPLAN Notices 12(3),* pp. 95-100, March 1977.

[17] P. M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems," *Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing,* Toulouse, pp. 129-134, June 1978.

[18] R. Milne and C. Strachey, *A Theory of Programming Language Semantics.* Chapman and Hall, London, 1976.

[19] B. Randell, P. A. Lee and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys, Vol. 10, No. 2,* pp. 123-165, June 1978.

[20] B. Randell, "System Structure for Fault Tolerance," *IEEE Transactions on Software Engineering, Vol. SE-1, No. 2,* pp. 220-232, 1975.

[21] D. L. Russell and M. J. Tiedeman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9: Ninth Annual International Symposium on Fault-Tolerant Computing,* Madison WI, pp. 106-109, June 1979.

[22] S. K. Shrivastava, "A Dependency, Commitment and Recovery Model for Atomic Actions," *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems,* Pittsburgh PA, pp.112-119, July 1982.

[23] S. K. Shrivastava and J-P. Banatre, "Reliable Resource Allocation Between Unreliable Processes," *IEEE Transactions on Software Engineering, Vol. SE-4, No. 3,* pp. 230-241, May 1978.

[24] A. Z. Spector and P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing," *Operating Systems Review, Vol. 17, No. 2,* pp. 18-35, April 1983.

[25] W. G. Wood, "A Decentralised Recovery Control Protocol," *Digest of Papers FTCS-11: Eleventh Annual International Conference on Fault-Tolerant Computing,* Portland, pp. 159-164, June 1981.

ORIGINAL PAGE IS
OF POOR QUALITY

APPENDIX

Resolution Algorithm and Mechanism for Planned Atomic Actions

We will assume a distributed system in which processes can exchange messages. Three kinds of message are employed:

1) Raise exception x in atomic action aa.

2) Acknowledge exception in atomic action aa.

3) Commit components involved in atomic action aa to invoke the exception handlers.

The message passing system includes time-out, check-sum, and other facilities to ensure reliable transmission. If the message passing system fails to transmit or receive a message for a process attempting recovery, an undefined exception is raised in that process. Each process, message, context, and exception can be uniquely identified. Some (unspecified) mechanism provides a mapping from an atomic action into the processes that are engaged in that atomic action.

The following distributed algorithm, executed by each process in an atomic action, implements the resolution mechanism:

```
(*When an exception is detected:*)

when receive("raise",x:exception; aa:atomic_action) or raised(x:exception; aa:atomic_action) do
      {
       (*Save current pending exception for broadcast condition.   *)
       previous_pending[aa] := pending[aa];

       (*Resolve exception within tree and save in 'pending'. *)
       if raised(x) then
             {
              pending[aa] := if node(x,exception_tree[aa]) then x
                             else universal_exception
             }
       else pending[aa] := root_smallest_subtree(exception_tree[aa],pending[aa],x )

       (*Check for an exception handler for the exception.*)
       if handler[aa,pending[aa]]=nil then pending[aa] := universal_exception

       (*Let the other processes know which exception is pending  *)
       (*in this process if a raised exception changes the pending  *)
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
(*exception or a raise message results in a new exception.    *)
if  pending[aa] <> x or
       (raised(x) and (previous_pending[aa] <> pending[aa]) then
    {
     broadcast("raise",pending[aa],aa) to other_processes[aa]
     replies_needed[aa] := number(other_processes[aa])
    }
else
    {
     replies_needed[aa] := 0
     (*Save acknowledgments until can suspend process.*)
     enqueue_ack("acknowledgement",aa,source_process)
    }

(*Finish any internal atomic actions.*)
if internal_atomic_action[aa].active then
    {
     (*If permitted abort internal atomic action.*)
     if internal_atomic_action[aa].aborts and
            not internal_atomic_action[aa].aborted then
            {
             (*Abort only on the first raised exception     *)
             (*in the containing atomic action.             *)
             internal_atomic_action[aa].aborted := true
             raise(abort,internal_atomic_action[aa])
            }
     (*Let the internal atomic action finish     *)
     (*using a co-routine resume.                *)
     resume_process
    }
else
    {
     (*If there are no internal atomic actions                  *)
     (*send any acknowledgments and suspend the process.        *)
     while queued_acks do send(dequeue_ack)
     suspend_process
    }
}
```

(*When a process returns from an internal atomic action*)

```
(*to a containing atomic action with a pending       *)
(*exception:                                         *)

when process_returns(aa:atomic_action) do
        if pending[aa] <> nil then
                {
                 (*Send any acknowledgements.*)
                 while queued_acks do send(dequeue_ack)
                 suspend_process
                }


(*When an acknowledgement is received for the      *)
(*last broadcast made, check to see if every       *)
(*acknowledgment has been received:                *)

when receive("acknowledgement",aa:atomic_action) do
        if acknowledge_last_broadcast and (replies_needed[aa] > 0) then
                {
                 replies_needed[aa] := replies_needed[aa]-1
                 if replies_needed[aa] = 0 then
                         broadcast("commit",aa) to other_processes[aa]
                execute(handler[aa,pending[aa]])
                }
        else ignore


(*When resolution has finished, one or more of the processes        *)
(*will have received a complete set of acknowledgments.             *)
(*These processes broadcast a commit, and every process             *)
(*which receives a commit may commence recovery.                    *)
(*Note that processes cannot commit until all internal              *)
(*atomic actions are terminated and acknowledgements made.          *)

when receive("commit",aa) do
        execute(handler[aa,pending[aa]])
```

Fig. A: A resolution mechanism algorithm.

The algorithm, shown in Fig. A, consists of four parts corresponding to (i) the detection of an exception, (ii) the completion of internal atomic actions, (iii) the receipt

ORIGINAL PAGE IS OF POOR QUALITY

of an acknowledgment and (iv) the receipt of a commit message. The algorithm terminates after one or more processes receive a complete set of replies to a broadcast. Each of these processes then broadcasts a "commit" message to all the other processes in the atomic action permitting them to begin recovery. Separate copies of the algorithm are instantiated for each of the processes involved in the atomic actions. Each copy of the algorithm has its own set of variables. The four parts of each instantiation of the algorithm are mutually exclusive. A measure of the complexity of the algorithm is the total number of messages required to establish exception handling. The minimum number of messages occurs when only one process detects an error and transmits an exception message to the other processes engaged in an atomic action. The minimum is:

$$3(n-1)$$

where n is the number of processes involved in the atomic action. The maximum number of messages required to establish recovery within a particular atomic action occurs when:

1 Every process detects an error and sends exception messages to the other processes in the atomic action concurrently. This contributes n(n-1) messages.

2 Every process receives one exception message and sends new exception messages to the other processes in the atomic action concurrently.

3 2) is repeated the maximum of n-2 times after which every exception has been received by every process. This contributes n(n-1)(n-2) messages.

4 Every process replies to the last n-1 exception messages. This contributes n(n-1) messages.

5 Every process broadcasts a commit. This contributes n(n-1) messages.

The maximum number of messages is:

$$n(n-1)+ n(n-1)(n-2)+ n(n-1)+ n(n-1)$$

or

$$n(n-1)(n+1).$$

This assumes that the height of the exception tree is at least n.

Although the resolution mechanism requires communication among the processes in an atomic action there is no overhead if an exception is not raised. Moreover, the overhead can be much reduced by centralizing the control of the resolution mechanism, minimizing the height of the exception tree (or number of exceptions), or minimizing the number of processes in each atomic action.

APPENDIX C

Fault Tolerance using Communicating Sequential Processes

To be presented at International Symposium
on Fault-Tolerant Computing

Kissimi, Florida
June, 1984

1. Introduction

Several practical techniques for the construction of fault-tolerant software have evolved in order to improve the reliability of computer systems [RAN78]. The aim of these techniques is to ensure that the system provides the intended service despite possible software (including software design) or hardware faults. The techniques depend upon two complementary approaches to fault-tolerance known as *forwards error recovery* and *backwards error recovery.* The two approaches complement one another and it has been suggested that both be used to provide more reliable software [RAN78, AND81, CRI82, CAM83].

Forwards error recovery aims to identify the error and, based on this knowledge, correct the system state containing the error [BES81, CRI83]. The approach requires accurate damage assessment and identification of the cause of error. *Exceptions and Exception Handlers* are a common mechanism used to provide forwards recovery [AND81, LIS82]. In contrast, *backwards error recovery* corrects the system state by restoring the system to a state which occurred prior to the manifestation of the fault. The *recovery block scheme* [RAN75] provides a system structure that supports backwards recovery. In the recovery block scheme, the system state is saved at various points called *recovery points.* If the system is later found to be in an inconsistent state, discovered by detecting an exception or applying an *acceptance test* to the state of the system, it may be restored to a state stored at a previous recovery point. The computation is repeated with a different algorithm called an *alternate.* If the system state after the alternate passes the acceptance test, then normal computation is resumed. Otherwise, the earlier state may be restored again and another alternate attempted. Eventually, if all the alternates for a given recovery point have been attempted and all have failed to satisfy the acceptance test, the system state is restored to an even earlier recovery point.

The recovery block scheme is well-suited for in a sequential program environment. The extension of this scheme for use with concurrent processes, where the the processes may share information, is not easy because of the possibility of error propagation between processes as a result of interprocess communication. If communication exchanges have not been co-ordinated with the establishment of recovery

ORIGINAL PAGE IS
OF POOR QUALITY

points, the discovery of an error may result in an uncontrolled rollback of many processes called the *domino effect* [RAN75].

The domino effect may not occur very often in practice and a chase protocol has been devised [MER78, WOO81] to compute the appropriate recovery points to which a group of communicating concurrent processes must be restored. Alternatively, the domino effect can be avoided by structuring the interactions between processes and the establishing of recovery points [KIM78, KIM80, RAN75]. If the actual recovery strategy must be computed at run-time, the scheme is dynamic [KIM78, KIM80]. If the recovery strategies for errors can be computed before execution, perhaps by a compiler, then the scheme is static [RAN75].

A language construct called a *conversation* [RAN75] has been proposed to provide a static backwards error recovery scheme. The conversation is an extension of the recovery block scheme to communicating processes. It isolates all the processes within a conversation from communications with other processes for the duration of the conversation. This limits the propagation of errors and eliminates the possibility of the domino effect. Each process may enter the conversation asynchronously and establish a recovery point. It should execute an acceptance test before it leaves the conversation. Processes leaving the conversation are delayed until every process within that conversation has passed its acceptance test. In this case, the conversation "succeeds". If *any* process fails its acceptance test, all processes must rollback to the start of the conversation and try their alternates. Several implementations of conversations have been described [SHR79, CAR83, AND83, KIM80].

Forwards error recovery in systems of communicating processes is discussed by Campbell and Randell in [CAM83]. A framework for exception handling is proposed that is based on the use of atomic actions.

In this paper, we use the framework provided by Campbell and Randell to support backwards and forwards error recovery in a system of Communicating Sequential Processes (CSP) [HOA78]. We present a construct called an S-Conversation which supports both backwards and forwards error recovery so that they may be used in a complementary way. The basic S-Conversation scheme can be implemented

ORIGINAL PAGE IS
OF POOR QUALITY

using only CSP primitives and the control for recovery is distributed over the processes taking part in communication.

2. Communicating Sequential Processes

CSP was proposed by Hoare as the basis for a concurrent programming language. CSP uses input/output commands for synchronization and communication. Message passing is synchronous though named *static channels*. An *output command* is of the form:

destination ! expression

where *destination* is the process name and *expression* is a simple or structured value. An *input command* has the form:

source ? target

where *source* is a process name and *target* is a simple or structured variable.

The commands *Ps ? target* in the process *Pr* and *Pr ! expression* in the process *Ps match* if the *target* and *expression* have the same type. Two processes communicate if they execute a matching pair of input/output commands. The result of executing a matching pair of commands is that the value of *expression* is assigned to *target*. There is no buffering and *Ps* or *Pr* must wait at the output or input command till the other process is ready to execute the matching command. After the communication, both processes proceed independently and concurrently. If *Ps* or *Pr* does not execute a matching command, the other process may wait forever. This inherent limitation of a synchronous message passing language makes detection of a so called "deserter" [KIM82] or dead process difficult.

Central to CSP is the use of Dijkstra's Guarded Commands [DIJ75]. A Guarded Command is of the form:

$$G \rightarrow C$$

where G is a guard and C is a command list. A guard is a list of boolean expressions which may be followed by an input command. Output commands may not appear in the guards. The command list may only be executed if every boolean expression in the list of the guard evaluates to "true". The alternative command has the form:

$$[\, G_1 \rightarrow C_1 \,\square\, G_2 \rightarrow C_2 \,\square \ldots \square\, G_n \rightarrow C_n \,]$$

and specifies the execution of exactly one of the constituent guarded commands. If none of the guards permit execution of a command list, the alternative command fails. If more than one guard allows a command list to be executed, a nondeterministic selection of one of the possible command lists is made.

A repetitive command of the form:

$$*\,[\text{ alternative command }]$$

specifies as many iterations as possible of the alternative command. The repetitive command terminates when all the guards fail.

Changes have been advocated to CSP [BER80, SIL78, SNE81] and many of them include the provision of output commands within the guards. We will use CSP with the facility of having output commands in the guards. We will assume that the language supports a primitive exception mechanism for a single process, although no such proposal exists in the original CSP paper.

3. The S-Conversation

We define a *Synchronized Conversation* (S-Conversation) as a distributed control structure which a group of processes may join or leave together in synchrony. While the processes are within the distributed control structure, they may communicate with one another but not with processes outside of the control structure. The S-conversation provides an "abstract atomic action" which is similar to, but more synchronized than, the atomic actions proposed by [CAM83]. The S-Conversation will be used as a framework within which backward and forward error recovery can be provided within CSP.

The aim of an S-Conversation is to provide a recoverable abstract atomic action within which processes may interact. For this it must have the following properties [KIM82, CAM83]:

1) A recovery line for backward error recovery. In the event of an error, the processes may be rolled back by restoring the states of the processes to the recovery points that were established at the recovery line. The S-Conversation provides a recovery line which is defined by the synchronized entry of all participating processes.

2) A test line for the processes. The test line is a set of diagnostic tests, one for each process, which is used to determine whether any errors have occurred during the S-Conversation before the processes leave the control structure synchronously. If errors have occurred, then those errors must be contained and repaired by recovery measures within the S-Conversation if the S-Conversation is to be reliable. In backwards error recovery, the diagnostic tests may include execution of "acceptance tests".

3) Recovery measures. If any process has errors, *all* processes must co-operatively invoke appropriate recovery measures. In the backward recovery scheme, if the acceptance test of any process is not satisfied, then each process must be rolled back to the recovery line and execute an alternate algorithm. In the forward recovery scheme, if a process detects an exception then all the processes in the S-Conversation should invoke their handlers for that exception.

4) Error confinement. The processes and their communications must be isolated inside the control structure from other processes and communications not in the control structure to prevent propagation of errors. The S-Conversation prevents *information smuggling*.

5) Recursive refinement of "abstract atomic actions" into other, more concrete, atomic actions. The S-Conversations may be strictly nested, one contained within another. The nested S-Conversation may only involve processes involved in the containing S-Conversation.

As a practical point, an implementation ought to detect and allow recovery from a *deserter process* or a process which dies [KIM82]. For example, a process expected to participate in an S-Conversation may not do so. This is a specially difficult problem to handle in a message passing system, since a process

ORIGINAL PAGE IS
OF POOR QUALITY

cannot unilaterally observe the state of another process (it can be done if communication is through shared data).

4. Error Recovery with S-Conversations

The S-Conversation can be used to support backwards and forwards error recovery for concurrent processes. For the purposes of explanation, we use the following syntax for an S-Conversation:

```
P1::[   ...
        S-Conv with ( P2, P3, ..., Pn )
                ...
                -- conversation of P1 with P2, P3, ..., Pn.
                ...
        end
        ...
    ]
```

Figure 4.1: An S-Conversation.

We require each process taking part in an S-Conversation to explicitly specify the S-Conversation construct. The syntax for an S-Conversation control construct includes a list of the other processes which will also be in the S-Conversation. Each of the processes taking part in the S-Conversation must agree as to the participant processes of the information exchange. The set of processes { P_1, P_2, ..., P_n } is called the C-Set of the S-Conversation. On entry to the S-Conversation, a run-time check establishes whether the C-Set for each participating process is the same.

Backwards error recovery can be programmed within an S-Conversation in the form indicated by the syntax shown in Fig. 4.2.

ORIGINAL PAGE IS
OF POOR QUALITY

```
P1::[   ...
        S-Conv with ( P2, P3, ..., Pn )
                ensure <acceptance test>
                by     <primary>
                else by <alternate>
                        ...
                else by <alternate>
                else error
        end
        ...
   ]
```

Figure 4.2: Backward Error Recovery using the S-Conversation.

The backward error recovery control construct implements a conversation [RAN75]. Some measure for recording the state of the variables of a process is executed as the process enters the S-Conversation. This is part of the implementation of the recovery line. Then, the primary algorithm is executed. At the end of the primary, the acceptance test is evaluated. If all the processes P_1, P_2, ..., P_n pass their respective acceptance tests, then the primary is completed by leaving the S-Conversation. If any of the processes fail their acceptance test or generate a run-time error during the primary, then each process in the S-Conversation is rolled back to the recovery line and the next alternate routine is executed. At the end of the alternate, the acceptance test is repeated. Eventually, either all the processes pass their respective acceptance tests or one or more of the processes have no alternates left to try. If any processes run out of alternates, *all* the processes fail and return an "error".

Since S-Conversations may be nested, the primaries and alternates may include further S-Conversations to provide error recovery and enhance reliability.

Forwards error recovery can be programmed within an S-Conversation in the form indicated by the syntax shown in Fig. 4.3.

```
P_1::[   ...
         S-Conv with ( P_2, P_3, ..., P_n )
                ensure no_exceptions
                by        <primary>
                on exception
                [         <exception_1>        -> <handler_1>
                          □ <exception_2>      -> <handler_2>
                          ...
                          □ <exception_n>      -> <handler_n>
                ]
                else error
         end
         ...
     ]
```

Figure 4.3: Forward Error Recovery using the S-Conversation.

The forwards error recovery is based on exception handling in asynchronous processes [CAM83]. A process entering the S-Conversation executes the primary routine. After completing the primary, the process waits at the test line for the completion of the other processes in the S-Conversation. If every process executed its primary without raising an exception, the processes can exit the S-Conversation. Otherwise, if an exception is detected during execution of the primary, then *all* the processes in the S-Conversation will be notified of the exception. Each process then executes the handler routine for that exception. Once again, each process will, when it finishes its handler, wait at a test line for the other processes to complete. If the handlers are all executed successfully without any further exceptions being detected, the S-Conversation can terminate normally. If exceptions are detected, then *all* the processes involved in the S-Conversation return an error.

Because of concurrency, several different exceptions might be raised simultaneously. In this paper, we assume that, whenever simultaneous exceptions are detected, they are resolved into a single exception which reflects the state of the S-Conversation by a mechanism based on a an approach similar to the exception resolution scheme discussed in [CAM83].

Forwards error recovery may be nested within backwards error recovery to take advantage of the complementary benefits of the two schemes as shown in Fig. 4.4.

```
P1::[   ...
        S-Conv with ( P2, P3, ..., Pn )
                ensure <acceptance test>
                by <primary>
                        on exception
                        [
                                <exception1>          -> <handler1>
                                □ <exception2>        -> <handlern>
                                ...
                                □ <exceptionn>        -> <handlern>
                        ]
                else by <alternate>
                ...
                else by error
        end
        ...
    ]
```

Figure 4.4: Forwards and Backwards Error Recovery using the S-Conversation.

If an exception is raised while the primary is executed, control changes from the primary to the appropriate exception handler. If the exception handler can recover successfully and complete the primary computation, the acceptance test of the backwards error recovery scheme is attempted. If the acceptance test is passed, the S-Conversation is terminated, otherwise a roll back is performed and the next alternate attempted.

However, the forward error recovery scheme may not be able to provide a successful recovery for a raised exception in the primary. In such cases, the exception will be passed to the backwards error recovery scheme and the process will be rolled back to the recovery line and then allowed to execute the next alternate. The nesting of S-Conversations allows many recovery schemes including the enhancement of an alternate with forwards error recovery.

ORIGINAL PAGE IS
OF POOR QUALITY

5. Recovery Primitives for S-Conversations

Three forms of error recovery - backwards, forwards, and combined - may be constructed using the S-Conversation control structure. In this section, we outline how these forms may be implemented using a common set of S-Conversation recovery primitives. The primitives support entry and exit into an S-Conversation and include a voting scheme. The primitives have CSP implementations which are described in a following section.

The basic S-Conversation primitives are shown in Fig. 5.1.

```
P_1::[   ...
         S-Conv (P_2, P_3, ..., P_n )
                 <code>
                 exit unless <exception>
                 <code>
                 exit unless <exception>
                 ...
         end
     ]
```

Figure 5.1: A Basic S-Conversation.

The body of the conversation includes "exit" statements which correspond to a test point within the test line. When the process reaches this point, it waits for other processes to reach their corresponding points in the S-Conversation. The "exception" is evaluated by a vote taken between all the processes and is null if the S-Conversation is successful. If the exception is null, then the S-Conversation has produced a result which meets the "ensure" specification and the exit statement terminates the control structure. Otherwise, the process continues within the S-Conversation and recovery measures are invoked. If none of the exits are taken, the S-Conversation completes when the process reaches the "end" statement.

In general, implementation of any of the forwards and backwards error recovery schemes will require the use of several exit primitives.

The S-Conversation primitives may be used to implement backwards error recovery using the scheme shown in Fig. 5.2.

```
P1::[    ...
         S-Conv (P2, P3, ..., Pn )
                <save state>
                <primary>
                exit unless <exception(<acceptance test>)>
                <restore state>
                <alternate>
                ...
                exit unless <exception(<acceptance test>)>
                <restore state>
                signal error
         end
         ...
    ]
```

Figure 5.2: Backward Recovery using S-Conversation Primitives.

The variables of the process are saved after entry to the S-Conversation. At the first test line, each process evaluates its acceptance test to detect an exception and this exception is compared with the result of the acceptance tests of the other processes. If any of the processes fails to satisfy its acceptance test, the exit statement will not terminate the construct. Instead, the variables of the process are restored to the values that were saved at the recovery point and the next alternate is executed. At the next test point, the acceptance test is again evaluated; this time using the values produced by the alternate. This continues until either the exit statement receives a vote indicating a null exception or the last alternate is attempted. The last alternate is used to return an exception to indicate that the S-Conversation has failed and the S-Conversation then completes.

ORIGINAL PAGE IS
OF POOR QUALITY

The S-Conversation primitives may be used to implement forwards error recovery using the scheme shown in Fig. 5.3.

```
P_1::[   ...
         S-Conv (P_2, P_3, ..., P_n )
                 <primary>
                 exit unless <exception>
                 [
                         exception_1      -> handler_1
                         □ exception_2    -> handler_2
                         ...
                         □ exception_n    -> handler_n
                 ]
                 exit unless <exception>
                 signal error
         end
         ...
  ]
```

Figure 5.3: Forward Recovery using S-Conversation Primitives.

The first test point after the primary detects whether any process detected an error. If no exceptions are raised, then the exit statement terminates the S-Conversation. Otherwise, the processes continue the S-Conversation by executing their handlers for the exception returned from the vote.

When the handler is completed, the processes invoke another test line. If this test line returns no exception, the exit statement terminates the S-Conversation. Otherwise, an error is returned and the S-Conversation completes.

Forwards and backwards error recovery schemes may be combined as shown in Fig. 5.4.

```
P1::[   ...
        S-Conv (P2, P3, ..., Pn )
                <save state>
                <primary>
                exit unless <exception(<acceptance test>)>
                [
                        exception1      -> handler1
                        □ exception2    -> handler2
                        ...
                        □ exceptionn    -> handlern
                ]
                exit unless <exception(<acceptance test>)>
                <restore state>
                <alternate>
                ...
                exit unless <exception(<acceptance test>)>
                <restore state>
                signal error
        end
        ...
    ]
```

Figure 5.4: Forwards and Backwards Recovery Combined.

Figure 5.4 is an implementation of the recovery scheme shown in Fig. 4.4. Having completed the primary, the process will wait at the first test point for the result of the vote on the acceptance test. If the acceptance test succeeds and no exceptions have been raised, then the S-Conversation terminates. If the acceptance test fails or an exception has been raised, the process attempts recovery by invoking the selected handler. A second test point reevaluates the acceptance test. This time, if an exception is raised within a handler or the acceptance test detects an exception, backwards error recovery is applied and the process executes the next alternate.

6. Implementation

A CSP-based implementation of the S-Conversation primitives is described in this section. The implementation uses only the CSP primitives for communication and synchronization between processes. The reliability of the recovery schemes is enhanced by compile and run-time checking.

A combination of compile and run-time checking is used to prevent information smuggling. A syntactic check ensures that a process participating in an S-Conversation only communicates to the other processes named in the C-Set of the S-Conversation. A further run-time check must be used to ensure that the C-Sets of the processes involved in a particular S-Conversation are the same.

The correct nesting of S-Conversations can be checked at compile-time by examining each process. Every process identifier which occurs in the C-Set of a nested S-Conversation must also occur in the C-Set of any enclosing S-Conversation.

The basic S-Conversation primitives are transformed into CSP primitives that implement the entry of processes into and the exit of processes from the S-Conversation control construct by a preprocessor. Both the entry and exit implementations involve a voting mechanism. For the purposes of implementation, we require the processes within an S-Conversation to have a static ordering (for example, we could use the lexicographic ordering of their identifiers).

6.1. S-Conversation Entry

Entry of a process into an S-Conversation requires synchronization and a C-Set consistency check. The consistency check uses a voting technique based on the Two Phase Commit protocol [GRA75]. Voting is implemented by passing a message up and down a chain of the processes attempting to enter the S-Conversation.

The processes whose identifiers are included in the C-Set of an S-Conversation are organized into a chain using their static ordering. In a vote, starting from the head of the chain, each process passes C-Set information to its successor. If the C-Set of any process does not agree with the information that the process receives, a C-Set exception is passed on. This ensures that the tail process will receive a C-Set exception if the C-Sets are not consistent. Next, the tail process returns the result of the vote back down the chain to the head. In this way, every process receives an exception if the C-Sets are inconsistent. If the C-Sets are inconsistent, the S-Conversation is aborted by each process aborting its local attempt to enter the S-Conversation. If a S-Conversation is aborted, each process could execute its primary; however, the recovery scheme of that S-Conversation cannot be used. If exceptions are detected the enclosing S-Conversation must perform the recovery.

The voting algorithm is shown in Fig. 6.1. Different algorithms are used for the head, middle and the tail of the chain. Since the chain is constructed using the static ordering of the processes, a compile-time algorithm can construct the voting scheme. We assume that process P_i is the predecessor of process P_{i+1}.

For the head of the chain (process P_1):

```
P_2 ! C_Set;
[       P_2 ? success ()  → proceed
      □ P_2 ? failure () → ABORT
]
```

For the middle of the chain (process P_i):

```
P_{i-1} ? C_Set ;
[       (C_Set = My_C_Set)    → P_{i+1} ! C_Set
      □ (C_Set ≠ My_C_Set) → P_{i+1} ! C_Set_Exception
]
[       P_{i+1} ? success ()  → P_{i-1} ! success ();
                                proceed
      □ P_{i+1} ? failure () → P_{i-1} ! failure ();
                                ABORT
]
```

For the tail (process P_n):

```
P_{n-1} ? C_Set;
[       (C_Set = My_C_Set)   → P_{n-1} ! success () ;
                               proceed;
      □ (C_Set ≠ My_C_Set) → P_{n-1} ! failure () ;
                               ABORT
]
```

Figure 6.1: Implementation of the entry into an S-Conversation.

The scheme has no mechanism to cope with the problem of a deserter process. If a process is in the C-Set of a set of processes taking part in an S-Conversation but it does not have an appropriate S-Conversation (a deserter process), then it will block entry into the S-Conversation because its neighbors in the S-Conversation voting chain will never be able to satisfy their I/O requests. A similar situation can arise if two processes have different C-Sets for the same S-Conversation. There appears to be no satisfactory solution to this problem unless a timeout mechanism is provided in CSP.

Suppose each process could start a preset timer when it tries to pass information to its successor process in the chain. If the process is unable to execute a matching input command within the set time, the preset timer could awaken it and it could then locally abort the S-Conversation. The same technique can be applied by each process when it expects input from its successor process in the chain. If one process aborts its S-Conversation, all the processes attempting to enter the S-Conversation will also eventually abort - either because they timeout or because they are informed of a C-set exception by a successor or predecessor process. For brevity, we do not consider the details of such schemes further in this paper.

6.2. The Exit Statement

The exit primitive is used to terminate an S-Conversation if it is successful. The exit primitive uses a chain-based voting scheme to decide whether an exception has been detected by any of the processes in the S-Conversation. If an exception is detected, all the processes in the S-Conversation must participate in recovery. Each process sends its successor process a result exception which reflects any exception that it may have detected as well as the exception passed to it by its predecessor. The final result is sent to each process in the S-Conversation by transmitting it back down the chain. The "value" of an exception is taken to be null if no exception occurred. The implementation scheme is shown in Fig. 6.2.

ORIGINAL PAGE IS
OF POOR QUALITY

For the head of the chain (process P_1):

```
P_2 ! my_exception;
[      (P_2 ? success ())→ exit
       □ (P_2 ? fail ())   → proceed
]
```

For the middle of the chain (process P_i):

```
P_{i-1} ? exception ;
P_{i+1} ! resolve(exception , my_exception) ;
[      (P_{i+1} ? success ()) → P_{i-1} ! success ();
                                exit
       □ (P_{i+1} ? fail ())  → P_{i-1} ! fail ();
                                proceed
]
```

For the tail (process P_n):

```
P_{n-1} ? exception ;
exception := resolve(exception,my_exception);
[      (exception = null)   → P_{n-1} ! success ()
       □ (exception ≠ null) → P_{n-1} ! fail ();
                              proceed
]
```

fig 6.2 : Translation of the exit statement

6.3. The Exception Mechanism

A process in a S-Conversation might raise an exception in a statement other than an acceptance test. In this situation, the processes in the S-Conversation should not continue with the normal processing. Instead all the processes should go to the exit statement and start the voting process. Such a circumstance also exists if an S-Conversation terminates abnormally with an error condition, in which case the recovery action of the enclosing S-Conversation should be invoked.

We require a mechanism which can notify processes inside an S-Conversation that an exception has been raised and change the control flow of the processes so they can terminate normal processing and start the voting process at the exit statement. Such a mechanism can be implemented if output commands are allowed in the CSP guard statement. We will briefly describe how this mechanism can be incorporated into the S-Conversation scheme using the CSP primitives.

If a process detects an exception, it transmits an exception message to the head process of the chain of processes. At each input or output statement, the head process checks for an exception message from any of the other processes in the S-Conversation. This can be done by transforming each input or output statement in the head process into an alternative command containing additional input statements. If a process informs it of an exception, it starts an exception vote at the next exit statement.

Each of the other processes check if their predecessor process has an exception vote to report at each input or output command. This can be done by transforming each input or output statement in the process into an alternative command that includes an input statement from the predecessor process. If informed of an exception, a process propagates the exception vote at the next exit statement. It transmits the vote in a similar manner to a regular vote and awaits the returned summary.

The exception vote uses the voting mechanism discussed previously. The head process initiates the voting and the exception is propagated along the chain. However, the head process is informed of an exception by one of the other processes in the S-Conversation.

If more than one process in the S-Conversation detects an exception, then only one process need be able to communicate its exception to the head process. The rest will be blocked in their attempt to communicate the exception. However, each of these blocked communications is in an alternative statement which also contains an input command from the predecessor process. Thus, any process which is blocked will eventually be activated by the receipt of an exception vote from its predecessor in the process chain. Once the blocked process receives this exception vote, it propagates the vote in the normal manner.

7. Conclusion

The paper proposes a technique for supporting backward and forward error recovery in a system of Communicating Sequential Processes. The technique uses a construct called an S-Conversation which coordinates an exchange of information between a group of processes. The S-Conversation supports forwards and backwards error recovery in a uniform manner. The control structure of a S-Conversation is distributed over the processes taking part in it. It is implemented using CSP primitives and supports local compile time and run-time checking to support reliable forwards and backwards error recovery. The number of communication messages needed to coordinate the S-conversation is O(n), where n is the number of processes taking part in the S-Conversation. The minimum number of communications needed is also O(n) since all processes must receive at least one message.

Although we have considered practical support for error recovery in concurrent systems, much further research and development is still required. We have not devised a simple scheme for detecting deserter processes in CSP. This seems to be a limitation of the synchronous message passing system which CSP employs. We have assumed that an exception mechanism can be supported for individual processes in CSP. We have not considered the real-time issues and non-termination of processes.

We believe that a structure like the S-Conversation should be used in concurrent languages to provide both backwards and forwards error recovery support. This would encourage the development of reliable concurrent applications. We have shown that both recovery techniques may be provided using a small number of uniformly applied S-Conversation primitives. We have demonstrated the practicality of using both schemes by devising a mechanism which can be transformed into CSP language primitives. This mechanism may be distributed over the processes engaged in the S-Conversation. The benefit of supporting error recovery by using programming constructs such as the S-Conversation requires detailed investigation.

References

[AND,81] Anderson, T. and P. A. Lee, *Fault Tolerance, Principles and Practice.* Prentice-Hall International, Englewood Cliffs NJ,1981.

[AND83] Anderson, T. and M. R, Moulding, *Dialogues for Recovery Coordination in Concurrent Systems.* Technical Report, Computing Laboratory, University of Newcastle upon Tyne, 1983.

[BES81] Best, E., F. Cristian, *Systematic Detection of Exception Occurrences.* Science of Computer Programming, Vol. 1, No. 1. North Holland Pub. Co., 1981,pp. 115-144.

[BER80] Bernstein, A. J., *Output Guards and Nondeterminism in Communicating Sequential Processes,* ACM TOPLAS, Vol 2, No 2, April 1980, pp 234-238.

[BUC83] Buckley, G. N., A. Silberschatz, *An Effective Implementation for the Generalized Input-Output Construct of CSP,* ACM TOPLAS, Vol 5, No 2, April 1983, pp 223-235.

[CAR83] R. H. Campbell, T. Anderson and B. Randell, Practical Fault Tolerant Software for Asynchronous Systems, *SAFECOM 83,* Cambridge, October 1983.

[CAM83] Campbell, R. H., B. Randell, *Error Recovery in Asynchronous Systems.* Tech rept no. UIUCDCS-R-83-1148, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[CRI82] Cristian, F., *Exception Handling and Software Fault Tolerance,* IEEE Transactions on Computers, Vol C-31, No 6, June 1982, pp 531-540.

[CRI83] Cristian, F., *Reasoning about Programs with Exceptions,* Digest of papers FTCS 13: Thirteenth international symposium on Fault Tolerant Computing, Milano, Italy, June 1983, pp 188-195.

[DIJ75] Dijkstra, E. W., *Guarded Commands, Nondeterminancy and Formal Derivation of Programs,* CACM, Vol 18, No 8, Aug 1975, pp 453-457.

[GRA78] Gray, J. N., *Notes on Database Operating Systems,* in Operating Systems: An Advanced Course, Vol 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, pp 393-481.

[HOA78] Hoare, C. A. R., *Communicating Sequential Processes,* CACM, Vol 21, No 8, Aug 1978, pp 666-677.

[KIM82] Kim, K. H., *Approaches to Mechanization of the Conversation Scheme based on Monitors,* IEEE, Transactions on Software Engineering, Vol SE-8, No 3, May 1982, pp 189-197.

[KIM78] Kim, K. H., *An approach to Programmer-Transparent Coordination of Recovering Parallel Processes and its Efficient Implementation Rules,* Proc. 1978 International Conf. on Parallel Processing, Aug 1978, pp 58-68.

[KIM80] Kim, K. H., *An Implementation of a Programmer-Transparent Scheme for Coordinating Concurrent Processes in Recovery,* Proceedings COMSAC80, 1980, pp 615-621.

[LIS82] Liskov, B., *On Linguistic Support for Distributed Programs.* IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, 203-210.

[MER78] Merlin, P. M., B. Randell, *State Restoration in Distributed Systems,* Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing, Toulouse, June 1978, pp 129-134.

[RAN75] Randell, B., *System structure for software fault tolerance,* IEEE Transactions on Software Engineering, Vol SE-1, No 2, June 1975, pp 220-232.

[RAN78] Randell, B., P. A. Lee and P. C. Treleaven, *Reliability Issues in Computing System Design.* ACM Computing Surveys, Vol. 10, No. 2, June 1978, 123-165.

[RUS79] Russell, D. L., M. J. Tiedeman, *Multiprocess Recovery using conversations,* Digest of Papers

APPENDIX D

The Concept of Atomic Actions in Concurrent Systems

Preliminary Thesis proposal of
Pankaj Jalote

The Concept of Atomic Actions in Concurrent Systems

Pankaj Jalote

Preliminary Thesis Proposal

1. Introduction

The concept of indivisibility of actions has been in use almost since the interrupt facility was introduced in computer hardware. The possibility of interrupts forced the designer to identify the primitive activities provided by the system which cannot be interfered with even by an interrupt. The facility of setting interrupts off was provided so that the systems programmer can make an operation which is not indivisibly executed by the hardware, or which consists of many indivisible hardware operations, indivisible by taking care of the only event - interrupt- which can interfere with the operation and destroy its property of indivisibility.

Though the term atomic action might suggest an action which should be indivisible and so preclude any concurrent activity, we are interested in the conceptual atomicity of the operation. This means that at the level of abstraction at which the operation is being performed, it should appear the atomic. That is, it should enjoy the properties of a primitive action, namely indivisibility, strict sequencing and non-interference. At a lower level of abstraction the different sub operations constituting the atomic operation might interleave with actions of other operations, the only restriction is that the interleaving and 'interference' at lower levels be such that the overall effect at the level of the operation is that the operation was performed atomically.

We have already mentioned that an atomic action may not be a primitive action actually executed atomically, but might be made from many different actions. This concept of building 'larger' atomic actions has also been in existence for a long time. The concept of defining a function is precisely that of defining a 'large' atomic actions in terms of smaller actions. This hierarchy can be as deep as desired.

So, in a sense this concept of atomic actions has been fundamental to our thinking, and is also the basis of the top down design methodology of sequential programs and functional notation. In sequential programming the concept is well established and the languages for constructing sequential programs have constructs such as procedures and functions to support this view. In sequential programs, since there is no interference between processes, atomicity is automatically guaranteed and so is never explicitly mentioned.

The situation changes when we consider concurrent systems, where more that one process might be interacting to perform a task. The problem becomes more complex because of the information exchange between the processes, and what was an easily implemented concept in sequential programming, becomes a difficult problem in the face of concurrency. The need for atomicity and the difficulty of implementing it in concurrent systems was first felt in the area of operating systems which led to the discovery of semaphores by Dijkstra Semaphores provided a mechanism by which a programmer could assure that a sequence of actions could be regarded as indivisible.

Since then the problem has reappeared in the context of databases as well as that of fault tolerance. Different techniques have been employed to solve the problem in these different contexts.

The aim of this paper is to provide an understanding of atomic actions, study the nature of atomic actions in different types of concurrent systems, and show that atomicity is indeed fundamental, and many different requirements which appear in different contexts have actually the same goal: to have a mechanism to ensure atomicity of operations. We also aim to show many other advantages which accrue from having such a construct. By doing this we will hopefully have convinced the reader of the usefulness of having some construct for supporting atomic actions, and the fundamental nature of the property of atomicity. This will hopefully provide enough justification to claim that the provision of atomic actions should be included in languages designed for concurrent systems, so that many of the existing problems can be treated uniformly.

2. Atomic Actions

An atomic action is an operation, possibly consisting of many steps performed by many different processors, such that it appears 'primitive' and indivisible to its environment. So, to the environment it is like a primitive operation which transforms the state of the system from one state to another without having any intermediate states, and has the properties of indivisibility, non-interference and strict sequencing.

This definition does not preclude the possibility of atomic action having a structure of its own, though it should not be visible to the environment. This allows atomic actions to be nested, and an atomic action to be composed of many, possibly concurrent, atomic actions. The visibility rules needed to preserve the property of atomicity at each level require that an atomic action be only aware of the actions which are its immediate children. Nested actions aid in decomposing activities in a modular fashion and have been proposed by others [8,18,19].

The definition of atomicity implies that no communication can take place across the boundary of the atomic action. In other words, inside an atomic action the processes performing the atomic action are not aware of the existence of other processes outside the action and the processes outside are not aware of the activity inside the atomic action. This restriction is necessary to ensure that the "internal state" of the atomic action does not become visible from outside the action, thereby destroying the property of indivisibility. This definition of atomic action implies the definition used in Anderson and Lee[2], and is similar to one proposed by Lomet[19].

Though atomic actions are defined from the point of view of the environment of the actions, and though an operation acquires the property of atomicity, when to its environment, it appears atomic, there are cases when an action can be inherently atomic. That is, there are situations when the structure of the action itself guarantees the property of atomicity, and the environment need not be considered. The two phase locking protocol[10] is an example of implementation providing inherent atomicity. Such actions can also be recognized by looking at their execution history[5,6]. In reality, since looking at the whole environment may be infeasible, the aim of implementations should be to control the computation in such a way that the execution of an action is inherently atomic. A language mechanism to support atomicity, will provide atomic actions which will be inherently atomic.

There is another view of atomic actions held by Liskov [18] and Davis[8]. They require that atomic actions should not only be indivisible, but should also be *recoverable*. This means that the after effect of an atomic action is all-or-nothing: either all the objects remain in their initial state or change to their final state. So, if a failure occurs it must be possible to either complete the action or restore all objects to their initial states.

We believe that indivisibility is fundamental but recoverability is not. Recoverability is a property which is not needed in all the systems and should be built using the primitive atomic actions,if desired.

The definition of atomic action says nothing about how the boundaries of atomic actions are defined. The atomic actions might be *planned atomic actions*, which has been implied so far. Planned atomic actions are atomic actions that have been planned during the design time of the system, and supported at run time. A language construct for atomic actions (with proper run time support) will lie in this catagory.

In contrast atomic actions might be *dynamically identified atomic actions*. This approach looks at the execution history of the program and finds the set of actions which satisfy the property of atomicity. While this approach is useful for modeling and understanding atomic actions[5,6,20], it does not provide the programmer with any mechanism to implement or specify atomicity. So, from the point of view of aiding in design of concurrent programs, such atomic actions are not very interesting. For the rest of the paper the term atomic action will mean planned atomic actions only.

3. Requirements for atomic actions

Any implementation of an atomic action must satisfy certain conditions. In this section we define those requirements. These are general requirements, and as we shall see in the next section the importance of different requirements may be different under different systems.

1) *Well defined boundaries :* Each atomic action should have start and end boundaries, and it should have two side boundaries. By side boundaries we mean that if there is more than one process taking part in the action then the side boundaries of the atomic action seperate the processes taking part in the atomic action from those which are not. The start and end boundaries might be spread over several processes. Together the boundaries enclose the amount of computation which has been specified to be atomic, and which the implementation should ensure has the property of indivisbility and atomicity.

2) *Information containment or indivisibility :* An atomic action must not receive information from or pass information to any activity outside the boundaries of the atomic action. Unless the information containment property is satisfied, the indivisibility of the atomic action cannot be guaranteed.

3) *Nesting :* Atomic actions should be allowed to be nested. This would permit an atomic action to be defined in terms of other nested atomic actions. Nesting allows modular refinement and structuring of atomic activities. Only strict nesting can be allowed (that is, no boundary of a nested atomic action should cross any boundary of the enclosing action), else information containment may be violated.

4) *Concurrency :* An implementation for atomic actions should allow maximal concurrency. An approach to provide atomicity is to let processes run sequentially, but this is overly restrictive. So, an implementation should allow maximum possible concurrency, while preserving the atomicity property.

5) *Robustness :* An implementation should be robust. We include the properties of fairness, deadlock freeness etc. under this catagory. This property, like the previous property, is a desired property rather than a strict, basic requirement.

4. Nature of atomic actions

So far we have been talking about atomic actions as a concept, without giving it any concrete form. The form of atomic actions and the problems which might be encountered during implementation might actually depend on the nature of concurrent system and the kind of atomicity needed. For example, the problems in a shared memory system in which only a single process takes part in the atomic action, are different from those in a message passing based system where multiple processes are taking part in the atomic action. In this section we catagorize atomic actions into two types and discuss, without proposing any implementation, the problems associated with these two types of atomic actions.

4.1. Single process atomic actions

When a single process specifies an operation, consisting of many steps of the same process, to be executed atomically, we call such atomic action as single process atomic action (SPAA). The main feature of SPAA is that the specification of the atomic action boundary is entirely in one process. We should point out here that it does not mean that only one process will finally perform all the computation inside the atomic action. Many different processes might be invoked to perform different operations which constitute the atomic action. But, the boundary is specified in one process only, and hence the name.

Having only one process specify the atomic action does not mean that the problem providing SPAAs is easy. It can be easily shown that guaranteeing atomicity of the basic computational steps inside the body of an SPAA does not ensure atomicity of the entire SPAA[6,10] if there are other active processes whose actions might interfere with the actions of the SPAA. Special measures have to be taken to ensure atomicity of the entire computation of the SPAA.

Serializability[4,24] has been accepted as the criteria for ensuring atomicity. Serializability requires that the implementation of SPAA should be such that the net effect of performing the computation of different atomic actions should be same as performing the actions serially in some order. The serial order to which the actual execution is equivalent is immaterial.

We would like to point out here that though often serializability is used as a synonym for atomic action, it is actually a property of atomic actions. For SPAAs showing that the implementation is such that the property of serializability holds ensures proper implementation.

Let us now look at SPAAs under the two major kind of distributed systems: shared memory systems and message passing systems.

Shared memory systems are those, as the name suggests, in which different processes share data through shared memory. That is, different processes might be accessing and updating the same memory location. Primary examples of shared memory systems are databases and monitors.

The problem of SPAAs in shared memory systems can be stated as follows: A process wants to perform certain operation on the shared data atomically. Due to the presence of processes sharing the data, its operation can be interfered with by another process. What is needed in this situation is some way for a process to say that it wants an operation to be atomic, and some implementation to provide atomicity. Many of the requirements are easily satisfied in a shared memory environment for SPAAs. The boundaries are easily specified, and nesting is quite often not needed, though proper nesting can easily be checked for because the body of the atomic action is contained in one process only.

This problem has occured in the area of operating systems, and is referred to as the problem of mutual exclusion [27], though the concept of mutual exclusion is somewhat more restrictive than SPAA. Semaphores and monitors[12] are different techniques devised to provide atomicity in a shared memory environment.

In the context of databases, the problem of providing SPAAs is called the problem of concurrency control. The unit of atomicity needed in databases is called a transaction. A transaction is a sequence of basic operations on the shared data. Since many transactions may be active concurrently and performing operations on the same data, data inconsistency can result. Provision has to be made to ensure that the entire computation of a transaction has the property of indivisibility. Many different solutions have been proposed[3,10,21,28]. The main criteria for atomicity in databases has been that different operations performed on the shared data should appear to have been done in sequence and without interleaving, that is, the serializability condition should be satisfied.

In message passed systems there is no shared memory and the processes communicate by sending and receiving messages. Each process has exclusive access to the data it owns and so the data is not prone to concurrent access from many processes. However, a process may request the owner process of the data to perform certain operation on the data. In essence the sender of the reqest message becomes the client and the receiver the server. Example of such systems are Distributed processes [11] and ARGUS Such systems are conceptually similar to shared memory system, except that the data is distributed, and the request for an operation goes through the process which owns the data. Moreover, the system has to further handle the problems which occur in message passing systems like lost and duplicate messages, remote host down etc. So, though the problem of providing atomic actions is more complex due to distributed data and multiple nodes, conceptually the problem of SPAAs is still the same. The atomic action is specified in one process only and serializability is once again the major criteria for atomicity. Examples of atomic action in such a system are described in [1,18,29].

4.2. Multiple process atomic actions

An atomic action need not necessarily be part of only one process. It may extend over many processes. If many processes together specify and take part in an atomic action we call it a multiple process atomic action (MPAA). In this case the boundary of the atomic action extends over all the processes taking part in the atomic action. The start and end boundaries are no longer as simply defined and identified as in SPAAs. The start and end boundaries will be the set of entry and exit points respectively, correspoinding to the entry into, and exit from the atomic action by the constituent processes of the atomic action. Each process may have many steps between its entry and exit points. The side boundaries are critical in MPAAs. In SPAA the side boundaries were defined simply since only one process was between the two side boundaries and everything else was outside, and the issue of side boudary did not ever arise. In a sense MPAAs are SPAAs with another dimension added. An example of MPAA would be the conversation construct[25].

First let us show that MPAA is not simply a collection of SPAAs.

Claim: If the computation between the entry and exit points of each process is assured atomicity, it would not guarantee atomicity of the entire MPAA.

The argument to show that the validity of the claim is quite simple. Since the computation of each process which lies inside MPAA is made into an SPAA, serializability of SPAAs is assured. Let us say that these SPAAs run sequentially (this will preserve serializability). Between two SPAAs another action, which does not belong inside MPAA can come and execute, and change or read the data, thereby setting up communication between the inside and outside of the MPAA. This will violate the containment requirement of atomic action.

Having support for SPAA is not enough to support MPAA. (Though the converse is true, because an SPAA is a special case of an MPAA.) Since many processes take part in an MPAA, the control of the atomic action is distributed and processes must now cooperate and jointly work to implement the atomic action. The main problem in implementing MPAA will lie in creating and supporting proper boundaries. and the primary criteria for atomicity becomes the requirement of information containment, that is, there should be no communication across the boundaries of the atomic action. (However, the actions will also be serializable)

The distributed control of MPAA introduces another problem, namely, the problem of deserter process [14], a problem which has no counterpart in SPAA. The deserter process problem occurs when a process which is understood to be a participant in an atomic action by other processes taking part in the action, does not take part in the action. This can result in endless wait by other constituent processes of the action.

An example of MPAA in a shared memory environment is given by Kim[14], which uses monitors. and an example of MPAA in a message passing system is given by Jalote and Campbell[13], which uses Communicating Sequential Processes for the message passing system.

5. Properties of atomic actions

In this section we will look at some of the properties of atomic actions. Many have already appeared in the literature in different contexts and with different names. We would like to stress that all these are the benefits which accrue from having atomic actions, even though many of these properties have been taken to mean atomic actions themselves.

1) *Mutual Exclusion:* This is the terminology used in the context of operating systems, specifying that when two operations in two different processes, operate on the same shared data then they should execute in mutual exclusion. That is, only one operation at a time should be operating on the shared data. Atomicity guarantees mutual exclusion. It does not mean that any two atomic actions will run (or need to run) in mutual exclusion, but the actions will always be non interfering, implying mutual exclusion where ever needed. Atomic actions provide a more general property than mutual exclusion. Mutual exclusion is often overly restrictive[19] and so leads to loss of concurrency. Atomic actions have no such restriction because of their generality.

2) *Serializability and data consistency:* This is the terminology used in the context of databases. The requirement is that concurrently executing transactions (a sequence of basic operations) should exe-

cute such that the result is as if the transactions had executed serially in some order. Atomicity implies serializability. If transactions are specified as atomic actions, then a correct implementation of atomic actions would ensure serializability. In this sense, all the concurrency control protocol aim to implement atomic actions.

3) *Fault tolerance:* Fault tolerance in distributed systems is another area where atomic actions are of great help. The aim of fault tolerant techniques is to ensure that the system provides the intended service despite possible faults. The techniques depend upon two complementary approaches to fault-tolerance known as *forwards error recovery* and *backwards error recovery*. Forwards error recovery aims to identify the error and, based on this knowledge, correct the system state containing the error. *Exceptions and Exception Handlers* are a common mechanism used to provide forwards recovery[2,17]. In contrast, backwards error recovery corrects the system state by restoring the system to a state which occurred prior to the manifestation of the fault. *Recovery block scheme*[25] is often used to structure the system to support backward recovery.

The problem of providing fault tolerance necessarily involves damage assessment and containment[26], and this problem becomes complex in concurrent systems. Atomic actions provide convenient structure to support fault tolerance in concurrent systems. For providing backward recovery in concurrent systems a structure called conversation[25] has been proposed. As it turns out, the structure conversation is essentially a MPAA with synchronized exit by all the processes taking part in it. For providing forwards error recovery and combined error recovery the use of atomic actions has been recognized and a framework to do so has been presented by Campbell and Randell[7].

4) *Deadlock Freeness:* If all processing is done using atomic actions a system of concurrent processes will remain deadlock free. However, the implementation of the atomic action may not be deadlock free, which might get the system in a state of deadlock. But, at the level of atomic actions there will be no deadlock. Deadlock is a property of implementation of atomicity. For example, two phase locking, which is an implementation of atomicity, may cause system deadlock. If the implementation of atomic action is deadlock free, we can be assured that the system is deadlock free.

5) *Proving correctness:* Almost all the techniques for proving correctness of parallel programs in a shared memory system, assume atomic action at some level[16,22,23]. The reason is that since parallel processes may interfere, some basic activity has to be identified which will be guaranteed to execute without interference, that is, execute atomically. Because of the interference freeness, assertions can be made easily about the behavior of the action. With this as a basis proofs of larger actions can then be built on top of it. But, some level of atomicity is required and needs to be identified for making assertions about the programs. It seems that if atomicity is provided at the language level whereby larger operations can be specified and guaranteed to be interference free, the proof of the system should simplify considerably. However, no formal work has been done along these lines and how much the provision of atomic actions simplify the problem of proving correctness is a research problem.

6) *Program structuring:* In designing concurrent programs for shared memory environment, some assumption is needed about the atomicity of operations. In[9] the atomic actions assumed are clearly stated. We believe, that provision of atomic actions will help the designer in designing parallel programs, because he can specify any activity as atomic and then be assured of interference freeness of that activity. Consequently, he can concentrate on designing the structure of the action itself. This would transfer some of the burden of designing parallel programs from the system designer to the language designer and implementor. We would like to mention again that this too is an area which needs more research before definite claims can be made.

6. Implementation comments

It is not the intent of this paper to propose implementation strategies to support atomic actions. However, we would like to discuss some general issues about implementation.

An implementation can be either static or dynamic. This issue is more pertinent in implementation of SPAAs. A static method implies that the serial order to which the final execution of SPAAs is equivalent is determined fixed once and then it never changes. An example is the timestamp technique of concurrency control [refs], in which the effective ordering of SPAAs is same as the ordering of their timestamps. In general static ordering will result in loss of concurrency.

Dynamic schemes, on the other hand, do not fix the effective ordering of actions statically. The ordering depends on the order in which the operations are performed. Such techniques have potential to support more concurrency. Locking protocols[10] and the Delay/Re-Read Protocol of Mickunas, Jalote and Campbell[21] are be examples of dynamic schemes.

An implementation may either be optimistic or pessimistic (or combined). A pessimistic approach assumes the worst case and acts in a preventive fashion so that atomicity is assured. A pessimistic approach often leads to loss of concurrency. Locking protocols are examples of the pessimistic approach.

An optimistic approach works on the assumption that interference between atomic actions is rare, and so takes no precausions against it. Usually they will involve some strategies to redo the computation in case atomicity is being violated. Kung and Robertson's[15] method is an example of this.

The two strategies can be combined to use the benefits of both. The Delay/Re-Read Protocol is an example of this.

7. Conclusion

In this paper we have looked at the concept of atomic actions. We believe that atomicity is a fundamental concept and natural to our thinking. The need of atomicity has been felt in different areas and different names have been given to the problem in different contexts.

If provisions exist to declare any operation atomic and if support exists for ensuring atomicity, many advantages will result. The problems of ensuring mutual exclusion and serializability will not exist any more. Solution to two major problems in the areas of operating systems and databases will be granted as an effect of having atomic actions, and the systems designer need not bother about those problems. Providing fault tolerance will simplify considerably. For providing fault tolerance in concurrent systems some notion of atomicity is required and fault tolerance can be provided using atomic actions with considerably less effort.

We also believe that the provision of atomic actions will help in proving correctness of programs and will aid in designing parallel programs.

Though the concept of atomicity has been in existence for a long time, it is only recently people have started understanding the nature of atomic actions and the generality of the concept. The possibilities of having atomic actions in programming languages is a recent idea too. As a result many consequences of having atomic actions as basic structures are still open areas for research. We are currently devising a design methodology for parallel programs using atomic actions. We are also looking into the problem of proving correctness of parallel programs using atomic actions.

Refrences

1. Allchin, J. E. and McKendry, M. S. *Synchronization and recovery of actions.* In: **Proceedings of symp. on principles of distributed computing.** ACM SIGACT-SIGOPS, Montreal, 1983, pp. 17-19.

2. Anderson, T. and Lee, P. A. **Fault Tolerance, Principles and Practice.** Prentice-Hall International, Englewood Cliffs NJ, 1981.

3. Bernstein, P. A. and Goodman, N. *Concurrency control in distributed database systems.* **ACM Computing Surveys** (June 1981) vol. 13, no. 2, pp. 185-221.

4. Bernstein, P. A., Shipman, D. W. and Wong, W. S. *Formal aspects of serializability in database concurrency control.* **IEEE Transactions on Software Engineering** (May 1979) vol. SE-5, no. 3, pp. 203-216.

5. Best, E. *Atomicity of activities.* In: **Lecture Notes in Computer Science, Vol 84,** Wilfred Brauer, ed. Springer-Verlag, New York, 1980, pp. 226-250.

6. Best, E. and Randell, B. *A formal model of atomicity in asynchronous systems.* **Acta Informatica** (1981) vol. 16, pp. 93-124.

7. Campbell, R. H. and Randell, B. "Error Recovery in Asynchronous Systems", UIUCDCS-R-83-1148, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

8. Davis, C. T. *Data processing spheres of control.* **IBM System Journal** (1978) vol. 17, no. 2, pp. 179-198.

9. Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M. *On-the-fly garbage collection: an exercise in cooperation.* **Communications of the ACM** (Nov. 1978) vol. 21, no. 11, pp. 966-975.

10. Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. *The notion of consistency and predicate locks in a database system.* **Communications of the ACM** (Nov 1976) pp. 624-633.

11. Hansen, Per Brinch. *Distributed processes: A concurrent programming concept.* **Communications of the ACM** (Nov. 1978) vol. 21, no. 11, pp. 934-941.

12. Hoare, C. A. R. *Monitors, an operating system structuring concept.* **Communications of the ACM** (Oct 1974) vol. 17, no. 10, pp. 549-557.

13. Jalote, P. and Campbell, R. H. *Fault tolerance using communicating sequential processes.* In: **Proceedings, 14th International Symposium on Fault Tolerant Computing**, IEEE, ed., Kissimie, Florida, 1984, pp. to-be.

14. Kim, K. H. *Approaches to Mechanization of the Conversation Scheme based on Monitors.* **IEEE, Transactions on Software Engineering** (May 1982) vol. SE-8, no. 3, pp. 189-197.

15. Kung, H. T. and Robertson, J. T. *On optimistic methods for concurrency control.* **ACM Transactions on Database Systems** (June 1981).

16. Lamport, L. *Proving correctness of multiprocess programs.* **IEEE Transactions on Software Engineering** (March 1977) vol. SE-3, no. 2, pp. 125-143.

17. Liskov, B. H. *On Linguistic Support for Distributed Programs.* (May 1982) vol. IEEE, no. Transactions, pp. 203-210.

18. Liskov, B. H. and Scheifler, R. *Guardians and actions: Linguistic support for robust, distributed programs.* **ACM TOPLAS** (July 1983) vol. 5, no. 3, pp. 381-404.

19. Lomet, D. B. *Process structuring, synchronization, and recovery using atomic actions.* **SIGPLAN notices (ACM)** (March 1977) vol. 12, no. 2, pp. 128-137.

20. Merlin, P. M. and Randell, B. *State Restoration in Distributed Systems.* In: **Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing.**, Toulouse, 1978, pp. 129-134.

21. Mickunas, M. D., Jalote, P. and Campbell, R. H. *The Delay/Re-Read protocol for concurrency control.* In: **Proceedings, First International Conference on Data Engineering.** IEEE, Los Angles, California, 1984.

22. Owicki, S. and Gries, D. *An axiomatic proof technique for parallel programs.* **Acta Informatica** (1976) vol. 6, pp. 319-340.

23. —. *Verifying properties of parallel programs: an axiomatic approach.* **Communications of the ACM** (May 1976) vol. 19, no. 5, pp. 279-285.

24. Papadimitriou, C. H. *The serializability of concurrent database updates.* **Journal of the ACM** (Oct 1979) pp. 631-653.

25. Randell, B. *System structure for software fault tolerance.* **IEEE Transactions on Software Engineering** (June 1975) vol. SE-1, no. 2, pp. 220-232.

26. Randell, B., Lee, P. A. and Treleaven, P. C. *Reliability Issues in Computing System Design.* **ACM Computing Surveys** (June 1978) vol. 10, no. 2, pp. 123-165.

27. Shaw, A. C. **The logical design of operating systems.** Prentice-Hall, Englewood Cliffs, N.J., 1974.

28. Silberschatz, A. and Kedem, Z. M. *A family of locking protocols for database systems that are modeled as directed graphs.* **IEEE Transactions on Software Engineering** (Nov

1982) pp. 558-862.

29. Weihl, W. and Liskov, B. *Specification and implementation of resilient, atomic data types.* **SIGPLAN notices** (June 1983) vol. 18, no. 6, pp. 53-64.

APPENDIX E

Performance Measurements on UNIX United

UNIX UNITED PERFORMANCE MEASUREMENTS

The Newcastle Connection is a software package that may be used to combine networked Version 7 UNIX systems into a distributed system that supports parallelism, remote resource access and communications. UNIX United is functionally indistinguishable from a single processor UNIX system. Thus users do not need to know about interprocessor communication and network protocols to program distributed applications. Instead, they may construct such systems by using the standard UNIX facilities for file and device access, I/O, interprocess communication, foreground and background processing, and protection.

We have modified the Newcastle Connection to allow Berkeley 4.1a UNIX systems to be combined into a UNIX United system. We are porting the software to Berkeley 4.2. Other project work (not NASA funded) will incorporate several 68000-based workstations into the united system including IBM 9000s running V7 and XENIX®.

The Connection intercepts the kernel calls of user processes and maps the calls into local and remote operations. The Berkeley kernel extends the V7 kernel and hence requires extensions to the Connection. Communication between Berkeley UNIX United systems is implemented by adapting the Connection network interface to use the Berkeley TCP/IP network protocols. Eventually we intend to improve the efficiency of the Connection by simplifying its interface to the TCP/IP protocol software. Other UNIX systems may be included within a UNIX United system composed of Berkeley systems although they may not be able to use all of the Berkeley features remotely.

The current implementation of the distributed system has permitted us to make some preliminary evaluations and performance measurements of the system. These evaluations have allowed us to improve the performance of the Connection. We are currently examining how to provide resource allocation and load balancing facilities based on the UNIX United approach. We believe these facilities can be readily implemented through the use of an additional layer imposed between the Connection and user processes.

Performance:

We have conducted performance tests based on running readily available UNIX benchmark programs and on transfering a single file. The tests were conducted on lightly loaded VAX 750 computers running 4.1BSD. Each VAX has 2 Mbyte of main memory and one CDC 80 Mbyte disk drive connected to the UNIBUS.

The first test copies a two megabyte file. The timings below provide a rough comparison between the Berkeley rcp copy command and both the ordinary UNIX copy and the UNIX copy implemented with the Newcastle Connection.

Comparison: 2 Megabyte File Transfer

cmd	local to local	local to remote
rcp	1.2u, 18.7s, 1:45	0.8u, 49.9s, 1:08
cp	1.0u, 18.2s, 1:46	(doesn't apply)
nc-cp	2.1u, 18.5s, 1:45	5.6u, 52.6s, 2:06

Each copy transfered 2000 1 kbyte blocks. The UNIX copy command makes 4000 read and write operations at an average speed of 26 milliseconds per operation. The UNIX United remote copy averaged 34 milliseconds per remote operation.

The second set of performance measurements was made with a small set of test routines that were published in BYTE magazine. These measurements show the performance for compiling and executing code in both the single VAX and in the networked VAX environments. Compiling takes longer in the UNIX United environment because each program must be linked to the Newcastle Connection. However, the differences between execution times in the UNIX United and ordinary environments are so small as to be within experimental measurement error. In the UNIX United measurements below, the routine iotest performed 500 read and 500 write operations using a remote file. These were random operations using random length data. A further 65000 write operations were performed to set up the file in the same test. Each I/O operation in the local test takes an average of 1.7 milliseconds. This is a lot faster than the I/O operations performed doing the copy because the buffer cache for the file eliminates

ORIGINAL PAGE IS OF POOR QUALITY

most of the overhead. Each I/O operation in the remote test takes an average of 24 milliseconds. (The length of the records being read and written vary in this test and result in a different access overhead than the earlier copy test.)

A comparison of the data shows that the single machine UNIX benefits greatly from the use of the local file cache. A more detailed analysis of the performance of the UNIX United system reveals that the majority of the time spent in communication was in the TCP/IP and hardware transmission, not in the Newcastle Connection code.

		VAX 11/750		NC_VAX11/750	
cc eros.c					
	real	5,	5	10,	12
	user	2.3,	2.6	3.7,	3.8
	sys	1.6,	1.8	2.7,	3.3
execution					
	real	5,	4	6,	5
	user	4.4,	4.1	4.5,	4.1
	sys	0.2,	0.2	0.3,	0.3
cc fibo.c					
	real	5,	5	12,	12
	user	1.9,	2.3	3.3,	3.5
	sys	1.6,	1.5	2.7,	3.1
execution					
	real	6,	5	6,	5
	user	4.8,	4.7	5.0,	5.2
	sys	0.6,	0.2	0.3,	0.3
cc floatpt.c					
	real	5,	5	12,	14
	user	2.5,	2.6	3.6,	3.8
	sys	1.5,	1.7	2.8,	3.2
execution					
	real	10,	8	10,	9
	user	8.7,	8.4	9.0,	8.4
	sys	0.6,	0.1	0.4,	0.4
cc iotest.c					
	real	9,	10	13,	15
	user	4.0,	4.8	5.3,	6.4
	sys	1.8,	2.0	3.2,	3.1
execution					
	real	115,	117	1589,	1604

	user	5.1,	4.5	139.9,	128.8
	sys	105.9,	110.2	540.6,	557.0
cc sort.c					
	real	10,	9	14,	16
	user	3.9,	4.4	5.2,	5.6
	sys	1.7,	1.9	2.8,	3.4
execution					
	real	60,	54	55,	51
	user	58.3,	52.4	52.1,	48.1
	sys	0.5,	0.5	0.6,	0.7

A further analysis of the iotest performance gave the following profile for remote access to a file:

```
I/O test, 500 random reads/writes.
-file opened for sequential writing
-normal termination after writing data file
-file opened for random reading and writing
-run complete
139.9u 540.6s 26:29 42% 12+20k 5+24io 10pf+0w

 %time  cumsecs  #call   ms/call          name

  50.0   320.27  67005      4.78          __send
  19.1   442.60  67005      1.83          __receive
   8.7   498.60 134042      0.42          __signal
   6.0   536.88                           _alarm
   3.3   557.95                           mcount
   2.9   576.20  67005      0.27          __netcall
   2.0   589.23  65500      0.20          __rwrite
   1.8   600.85  67005      0.17          __netrcv
   1.1   608.15  65501      0.11          _write
   1.1   615.12  67004      0.10          __rcall
   1.0   621.35  67005      0.09          __netsend
   0.8   626.53  67005      0.08          __buildsn
   0.7   630.70  67004      0.06          __rmt_fd
   0.6   634.25      1   3550.00          _main
   0.5   637.48  66000      0.05          _bmove
   0.3   639.45  67005      0.03          _setjmp
   0.0   639.62    500      0.33          __rread
   0.0   639.77   1000      0.15          _random
   0.0   639.88   1000      0.12          _lseek
   0.0   639.95   1000      0.07          __rlseek
   0.0   640.00      7      7.14          __stat
   0.0   640.03                           __doprnt
   0.0   640.05      1     16.67          __getNC
   0.0   640.07      2      8.33          __umask
   0.0   640.08    500      0.03          _read
   0.0   640.08      1      0.00          __creat
   0.0   640.08      2      0.00          __delfd
```

```
0.0   640.08    2    0.00      __findnn
0.0   640.08    1    0.00      __getegid
0.0   640.08    1    0.00      __geteuid
0.0   640.08    1    0.00      __getgid
0.0   640.08    1    0.00      __getid
0.0   640.08    2    0.00      __getpid
0.0   640.08    1    0.00      __getpug
0.0   640.08    1    0.00      __getuid
0.0   640.08    1    0.00      __ioctl
0.0   640.08    5    0.00      __ipack
0.0   640.08    3    0.00      __locate
0.0   640.08    3    0.00      __mkfd
0.0   640.08    1    0.00      __ncinit
0.0   640.08    2    0.00      __netatoi
0.0   640.08    1    0.00      __netinit
0.0   640.08    2    0.00      __netitoa
0.0   640.08    1    0.00      __netopen
0.0   640.08    1    0.00      __netrslt
0.0   640.08    1    0.00      __newsrv
0.0   640.08    2    0.00      __rclose
0.0   640.08    1    0.00      __rcreat
0.0   640.08    1    0.00      __ropen
0.0   640.08    4    0.00      __rservice
0.0   640.08    1    0.00      __socket
0.0   640.08    1    0.00      __socketaddr
0.0   640.08    1    0.00      __write
0.0   640.08    2    0.00      _close
0.0   640.08    2    0.00      _creat
0.0   640.08    1    0.00      _ioctl
0.0   640.08    1    0.00      _open
```

The commands labeled "_ _command" are original UNIX utilities. The Newcastle Connection software has names beginning with "_command". As can be seen from the statistics, approximately 69% of the time required to access a remote file is taken up by the send and receive primitives. This indicates that lightweight protocols and fast, responsive networking hardware is more critical to good performance than any major improvements to the Connection software and the remote procedure call scheme.

Last, we compared the size of the iotest program when it is used to access a remote resource to the size when it is compiled on standard UNIX. From our studies, this is close to the maximum additional space required by any process using remote access to networked machines. The results, shown below, demonstrate that the network facility is very inexpensive with respect to space. The additional space required by the iotest program to access remote files is 6k bytes of program and 1kbytes of data space.

A few extra bytes of control information are also required (bss) for the remote procedure call interface, and for the Newcastle Connection naming schemes. From the point of view of UNIX, it would be better to include this overhead in the kernel where it would be shared by all processes performing remote access.

```
Script started on Mon Feb 13 23:05:13 1984

i1% cc iotest.c -o normal
i2% cc iotest.c -o connected
i3% size normal connected

textdatabssdechex
51201024194080841f94normal
1126420482104154163c38connected

Script done on Mon Feb 13 23:06:20 1984
```

We believe that the remote procedure call scheme, using lightweight protocols, removes the need for having large amounts of TCP/IP protocol software in the 4.2 kernel. Such space considerations, although probably not relevant to the users of UNIX, is very relevant in the design of small embedded systems, and has been considered in the design of EOS and DPP.

APPENDIX F

Illinet - A 32 Mbits/sec. Local Area Network

1. Introduction

The rapid development in VLSI technology has made host computers and terminals smaller and cheaper. In recent years it has become rather common for an organization to have several computing systems with substantial processing and memory capacity operated and maintained within the same building or in several closely located buildings. These computing systems may each serve a wide range of simple and intelligent terminals. The need to share data, programs, processing power, and I/O facilities invariably makes it necessary for the computers and terminals to be interconnected in the form of a local area network. Indeed, many local area networks have been designed and implemented. Among the well known local area networks are Xerox ETHERNET [1], Bell SPIDER [2], and LCSNET [3]. These networks have been designed to provide low delay access via interactive terminals to host computers at relatively low cost per interconnection and with ease for network extension and reconfiguration. Since the effective link bandwidth in such a network is divided evenly among all terminals and hosts, it is often impossible to facilitate transfer of large files between host computers at high speeds required in many applications.

Many studies have shown that the performance of a local resource sharing network and distributed data base system depends critically on the communication bandwidth between hosts [4]. In particular, an effective resource sharing environment can be achieved only when wide band data links between hosts are available to allow file transfers at speeds near those of fast I/O devices in the hosts. ILLINET is a local area network designed to accomplish this goal. Its structure is similar to the Distributed Computing System (DCS) at the University of California, Irvine [5]. This paper describes ILLINET which has been designed and is currently being implemented in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

In section 2 the design objectives of ILLINET are discussed. These objectives impose several constraints on the network configuration and control structure. Section 3 gives an overview of ILLINET. In section 4 the link level data packet format is described together with the hardwired network access and link control protocols. Finally, the hardware architecture is presented in section 5.

2. Design Objectives

ILLINET will eventually connect several PDP-11's, a PRIME computer, and a network of microcomputers. These computers are operated and maintained by the Department and are used in a variety of real-time and batch processing applications. Currently they are already interconnected via 9600 baud lines in a star configuration to provide access to 50-60 simple terminals. All of these computers are located within one building although ILLINET is designed to allow interbuilding connections. It is envisioned that one of the nodes on ILLINET will be a PDP-11 which will serve as a gateway to the main campus computing facility.

The primary purpose for the design and implementation of ILLINET is to enhance existing computation facilities so that the resultant computer network will support effectively a variety of research activities in the areas of distributed operating systems, distributed data base systems and file servers. In order to assure that transmission links between the nodes and link-control level protocols will not be the bottleneck in interprocessor communication and data flow, it was decided that ILLINET is to be constructed using the latest cost effective technology. The transmission medium used in ILLINET is fiber optics because of its ability to support high bit rates and allow reasonable interfaces. A link bandwidth of 32 Mbits per second is achieved with the use of ECL circuits. Since most of the network access and link control protocol functions are implemented in hardware, nearly all this bandwidth will be available for interprocessor communication.

The need to avoid the difficult task of providing bidirectional signal transmission and proper termination of the optical fibers dictated that ILLINET be a ring network. The packet switching discipline and distributed network control structure are used. Because of the high data link bandwidth and the relative short loop delay in ILLINET, it is not necessary that the most efficient network access control scheme be used. The version of token control scheme implemented in ILLINET is described in sections 3 and 5. It is similar to the scheme used in DCS. It will undoubtedly provide sufficiently low access delay and high network throughput.

In order to support high-level process communication in broadcast mode and to allow transparent transfer of destination process from one node to another, associative addressing is used in ILLINET. Address recognition hardware and link control protocols are both designed to support efficient broadcast communication in the network.

3. Network Overview

The configuration of ILLINET is described in Figure 1. It contains no central controller or primary station to carry out clock synchronization and access control functions. In each of the ring adaptors (RA) on the ring, there is a 16-bit active data path (hereafter referred to as front-end window) between the optical receiver and transmitter in the front end. More specifically, a RA functions as a repeater which retransmits the incoming data stream. The portion of the data stream appearing in the front-end window may be examined by the RA.

A host can gain access to the network via the ring adaptor attached to it. To each of the hosts on ILLINET, the network functions as a packet-switched network. To send a message, the host segments the message into network packets of a maximum size of 4K bits. Each packet is delivered individually to the RA where it is stored in one of the

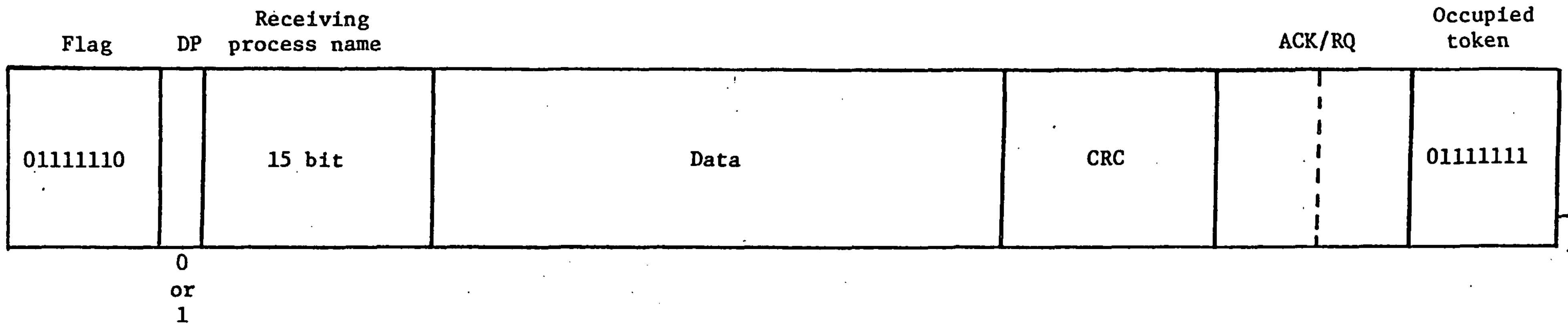

Flag
DP
Receiving process name
ACK/RQ
Occupied token
01111110
15 bit
Data
CRC
01111111
0 or 1

Figure 2

ORIGINAL PAGE IS
OF POOR QUALITY

output buffers. The completion of the loading of the data packet into the output buffer is acknowledged by an interrupt sent by the RA to the host. The host in turn can signal the RA to commence accessing the network and transmitting the data packet. The transmission of the data packet is then carried out under the control of the RA without host intervention. Under normal operating conditions, the data packets will be delivered to the destination in the order in which they are sent from the host to the RA, and duplicate and lost packets will not occur. However, reliable sequenced delivery is not guaranteed. Mechanisms to assure reliable datagram delivery and message sequencing and reassembly are carried out by the hosts.

The RA monitors the data stream passing through its front-end window at all times. When there is a packet to be delivered, the RA removes the access control token (01111111) from the ring when the token appears at its front-end window. There is only one control token in the ring. When the RA receives the token, it is allowed to transmit one data packet. The format of the data packet is shown in Figure 2. Besides the receiving process name there are CRC check, duplicate mark and acknowledge/repeat request fields. The data packet is retransmitted until positive acknowledgements are received from all RA's serving active processes whose names match the receiving process name in the packet header. (We will return to discuss the acknowledgment and repeat request features in the next section.) The use of this stop-and-wait ARQ scheme simplifies the host-to-host synchronization. Since the bandwidth

ORIGINAL PAGE IS
OF POOR QUALITY

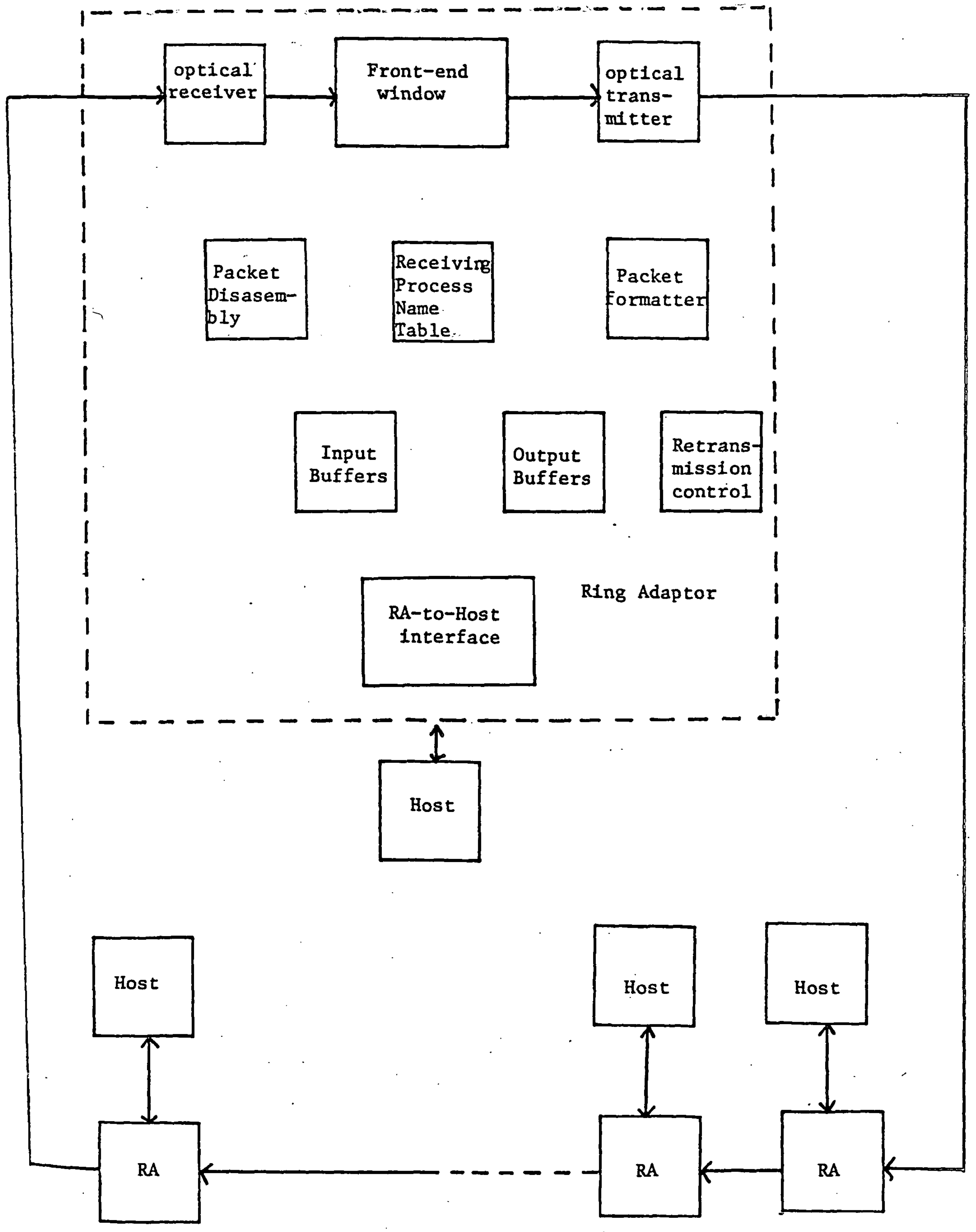


Figure 1

of the host-to-RA interface is significantly lower than the network bandwidth, the host-to-host throughput will not be limited by its use [6]. Furthermore, since there are two output buffers in the RA, network access for transmission from one buffer over the network can be carried out while the host loads the other output buffer. Thus, the speed of large file transfer between hosts will be limited primarily by the bandwidth of the host-to-RA interfaces.

In order to facilitate dynamic renaming and broadcast mode communication, a high-speed static RAM is provided in each RA to store the local active process name table as suggested in [7]. This table is updated by the host. When a data packet passes the front-end window of a RA, the RA checks the receiving process name field to determine whether it matches any of the names in its process name table. The data packet is copied into one of the 16 input buffers as it is simultaneously subjected to CRC checks. The data packet is kept in the input buffer only when there is a process name match, the data packet is free of error, and there is an input buffer available for its storage. In this case, the RA interrupts the host to inform it of the reception of the data packet. As the data packet passes through its front-end window, the RA makes comment in the acknowledgment/repeat request field. Such comments serve as acknowledgments to the sending RA.

Within the ring only the data path between the optical receiver and transmitter inside the sending RA is open. Hence, under normal operating conditions, the sending RA will remove the data packet when it returns to the optical receiver. By checking the contents of the acknowledgment/repeat request field in the returned data packet, the sending RA can decide immediately whether retransmission of the data packet is warranted. Either when the data packet transmission is completed successfully or is aborted after retransmission a maximum number of times, the sending RA releases the token and interrupts the host. By checking the status of the RA, the sending host can determine whether the transmission of the data packet is successful. If the other output buffer is nonempty and if the transmission of the previous data packet is successful, the host may signal the RA to commence network access again. On the other hand, if the delivery of the data packet fails, the host may ask the RA to attempt retransmission again or to invoke error diagnosis process. Thus, the sending RA is guaranteed the use of the data link for the delivery of both the data packet and the associated acknowledgment.

4. Packet Format and Link Control Protocols

The link level data packet format is shown in Figure 2. The data field is sandwiched between the packet header and trailing control fields. The header consists of the flag, "01111110," marking the beginning of a data packet, duplicate mark (DP), and the receiving process name field. The sending process name, packet sequence number and higher-level control information are considered here as parts of the data field. The trailing control fields consist of the cyclic redundant check code (CRC), the acknowledgment/repeat request (ACK/RQ) field, and the occupied token "01111111",* marking the end of the data packet. The receiving process name and the data are supplied by the host. The other fields are generated by the sending RA.

We note that the data packet format is similar to that in HDLC. To achieve data transparency a zero is inserted following every occurrence of 5 contiguous 1's in the data steam between the flags and the occupied token as in HDLC. The flag and the token are the only control fields containing more than 5 1's and hence can be uniquely identified at link level. Before the zero insertion the data field is n x 16 bits long for some n between 0 and 255. The 16-bit CRC code specified by the generating polynomial $x^{16} + x^{12} + x^{5} + 5$ is used for detecting errors in all bits between the flag and the ACK/RQ field.

A data packet is marked as a duplicate by the sending RA with its DP set to 1. A RA can check the first 16 bits (after zero deletion) following the flag to determine if the packet is intended for some local process and whether the data packet is a duplicate one. The last 8 bit field before the occupied token is the ACK/RQ field. When a data packet leaves the sending RA, its ACK/RQ field is reset to off to mean negative

*The bit pattern representing the occupied token is the same as that used to represent the control token. That a token is occupied (and, therefore, is not trapped by a RA which is waiting to obtain the token) is signified by this pattern following a matching flag.

acknowledgment and no repeat request. As the data packet passes through its front end, each RA on the ring may acknowledge whether the data packet is properly received by marking its comment in the ACK/RQ field. A RA sets the ACK field if the receiving process name matches the name of a local process name and if the data packet is copied and stored in its input buffer ready to be delivered to the local process. The RQ field is set when there is a process name match. However, either due to error detected in the data packet or due to input buffer overflow, the data packet is not correctly copied into the input buffer. Thus, the RA may request the data packet be retransmitted.

The operations of the sending RA is described by the flowchart in Figure 3. Before transmitting a data packet, the acknowledgment state of the sending RA and the number of retransmissions count are initially reset to zero. When the data packet is being transmitted for the first time, the duplicate mark is set to 0. As the data packet makes a round trip around the ring appropriate comments are collected in the ACK/RQ field from all RA's on the ring. By scanning the ACK field, the sending RA may determine whether the ACK field is set (meaning that some RA made a positive acknowledgment). If the ACK field is set, the acknowledgment state of the sending RA is set to 1. The repeat request field is set if any RA made a repeat request. The sending RA will immediately retransmit the data packet in this case. However, this time the DP bit is set to 1 to mark the data packet as a duplicate. If, on the other hand, the RQ field is found to be off when the data packet returns to the optical

i:	transmission account
I:	the number of maximum allowable transmission attempts
S:	Acknowledgement State
DP:	duplicate mark
ACK:	Acknowledgement
RQ:	Repeat Request

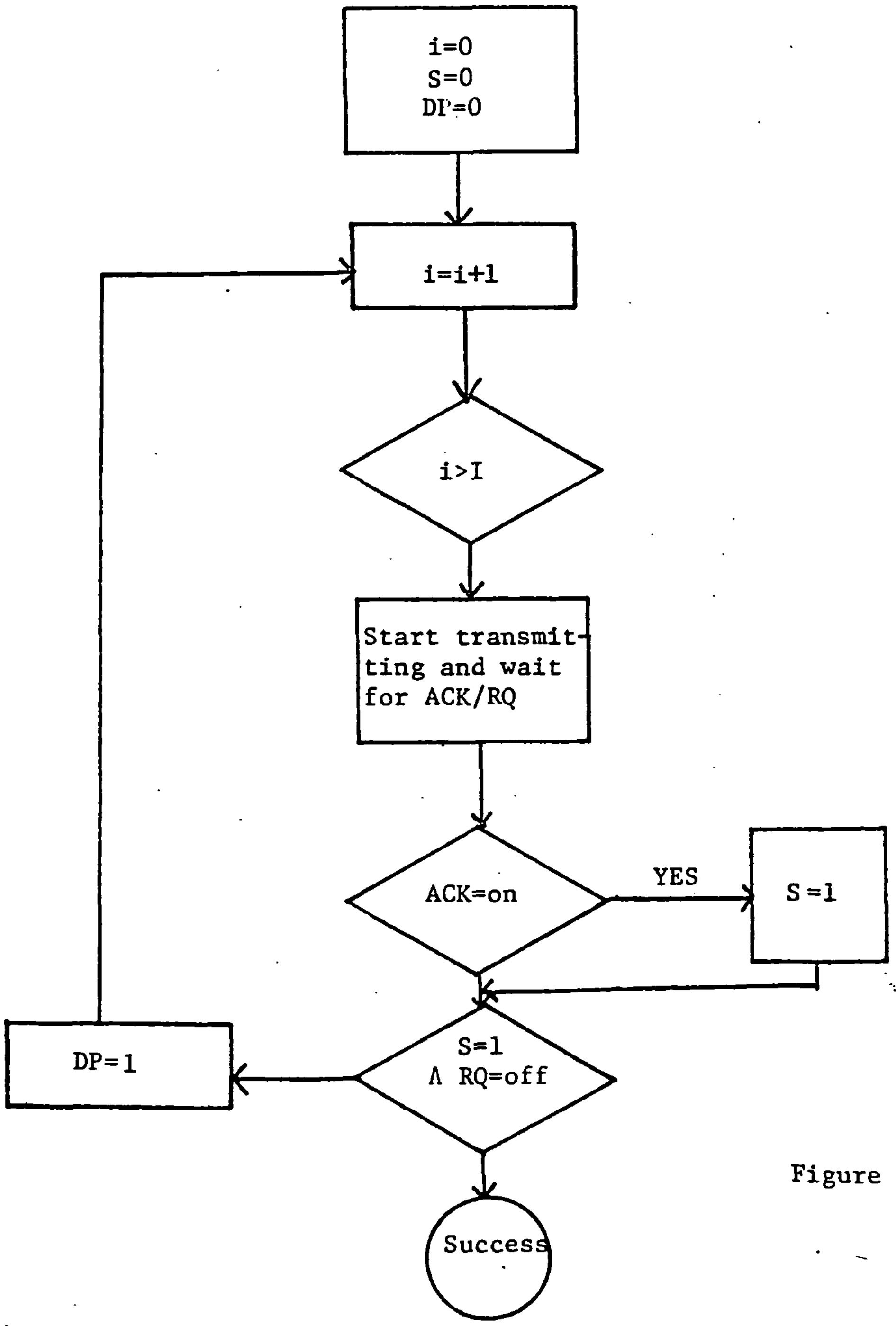
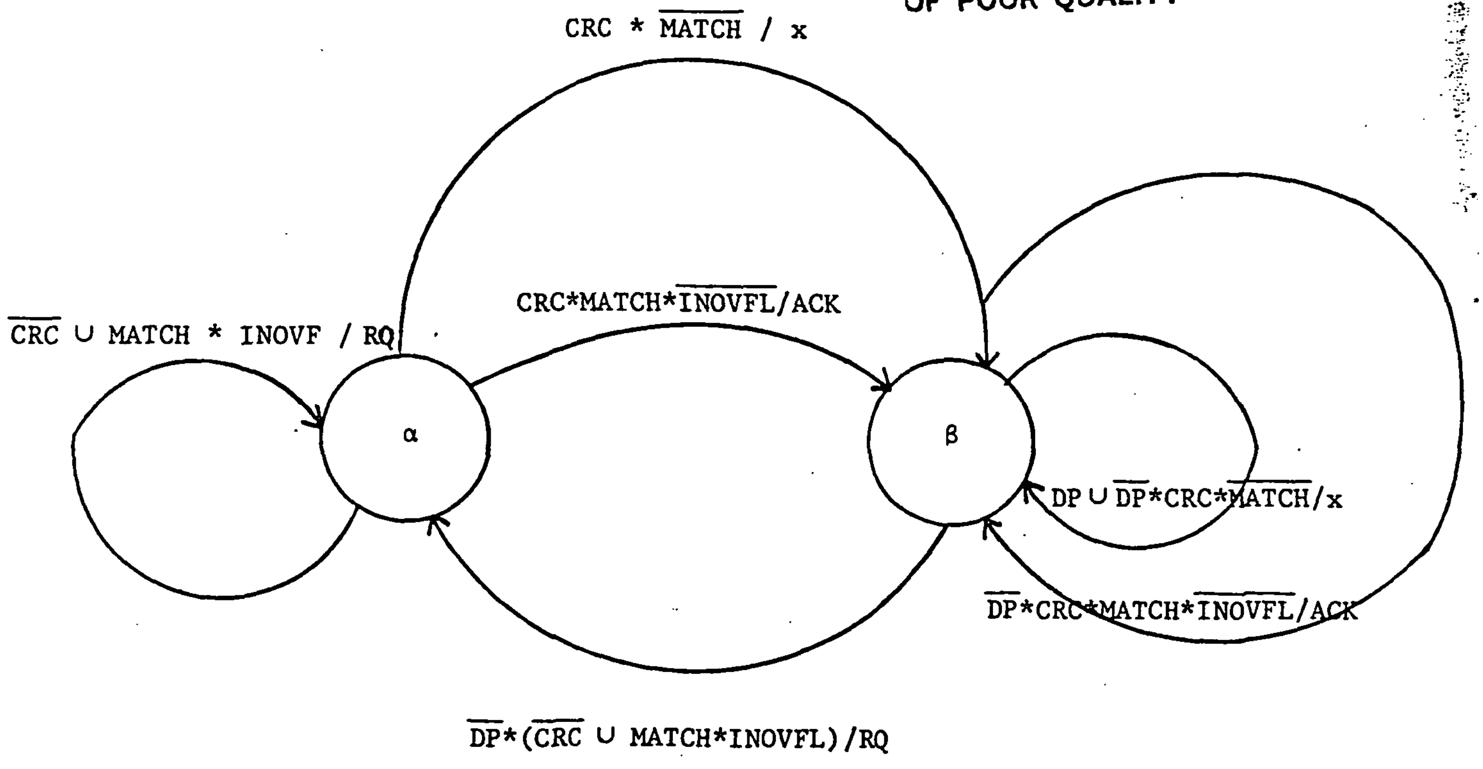


Figure 3

receiver in the sending RA, the transmission is considered completed. We note that the acknowledgment and the repeat request fields in the returned data packet not set by any RA will be interpreted by the sending RA that the receiving process name does not match the name of any active processes on the ring. In this case, the sending RA immediately retransmits the data packet. Since this data packet has not been received by any RA, it is not marked as a duplicate.

The operation of a RA which is not transmitting is described by the state transition diagram in Figure 4. Such a RA is in one of two states, α or β. When the DP in a data packet is 0, any RA may copy the data packet and make comment in the ACK/RQ field. Once a RA receives a data packet and stores it in an input buffer, it writes a positive acknowledgment in the ACK/RQ field of the data packet and enters state β. As this data packet reaches the sending RA, its acknowledgment state will be set to 1. Hence, during subsequent retransmission of the data packet, the DP is set to 1. While it is in state β, the RA is inhibited to make comment in any data packet marked as duplicate.

A RA enters state α when it makes a repeat request comment in the RQ field of the last data packet passing through its front-end window. When the duplicate mark in the data packet is set to 1, RA copies the data packet into its input buffer and makes positive acknowledgment in the ACK/RQ field only if it is in state α. Thus, reception of duplicate packets is prevented except in the relatively rare cases when noise


$CRC * \overline{MATCH} / x$
$\overline{CRC} \cup MATCH * INOVF / RQ$
$CRC*MATCH*\overline{INOVFL}/ACK$
α
β
$DP \cup \overline{DP}*CRC*\overline{MATCH}/x$
$\overline{DP}*CRC*MATCH*\overline{INOVFL}/ACK$
$\overline{DP}*(\overline{CRC} \cup MATCH*INOVFL)/RQ$

Legend: A/B

A: condition

CRC:	CRC error code o.k.
DP:	duplicate mark
MATCH:	receiving process name of the data packet matches with that of some local process.
INOVFL:	input buffer overflow

B: action

ACK:	positive acknowledgement
RQ:	repeat request
x:	no comment

Figure 4

causes messages to be garbled on more than one link in the network.

To summarize, a RA which is not transmitting will set the ACK field of the data packet passing through its front-end window and thus make a positive acknowledgment of its reception if (1) it is in α state, or it is in β state, but the DP of the data packet is 0, (2) the receiving process name in the data packet matches some process name in the receiving RA, (3) there are free input buffers, and (4) the CRC check detected no error in the data packet. Similarly, it will make a repeat request if it is in state α, or it is in state β, the DP of data packet is 0, and one of the following conditions is true: (1) the CRC check found the data packet to be errorous, or (2) there is no free input buffer and the receiving process name of the data packet matches with that of some local process.

We note that there is no need to initialize the RA to be in α or β state. Being self-synchronized, the RA should function correctly even if some RA's are in state α and some RA's are in state β at the time when the transmission of any data packet commences.

4. Error Recovery

In the two nodes that have been implemented to date, error recovery hardware is not included. Because of the limited knowledge in the failure characteristics of the type of networks such as ILLINET, it was decided to postpone the implementation of these hardwares. Instead, network error recovery functions are carried out by the hosts. However, time-out and interrupt circuits are included in each of the RA's for the detection of malfunctions in the RA or networks. For example, when a RA has a data packet to be sent but has waited for a long time for the control token, an interrupt is sent to the host when a preset time-out period expires to alert the host possible network malfunctions and invoke recovery procedure. Similarly, if after a RA caught the control token and transmitted a data packet but the occupied token at the end of the data packet does not return after a maximum loop delay or if the transmission lasted too long a period of time, appropriate interrupt signals are sent to the host. Status bits within the RA are provided to aid the host in its diagnosis to pinpoint the cause of network malfunction. The input buffer and output buffer memory modules are completely independent. Therefore, it is possible to support echo transmission mode. In this mode, a sending RA stores the data packet transmitted from its own output buffer when the packet returns from the network. It is also possible for a host to separate the RA from the network. In this case, a data packet may be transmitted directly from the output buffer to the input buffer of the RA. Thus, individual RA's can be tested independently making isolation of malfunctioned RA a relatively easy task.

Hardware for recovery from error conditions involving the token is included in our design. Under normal operating conditions there is only one control token circulating around the ring. Failure or transient noises in both RA's or the link may cause the token to be lost or duplicated. We refer to these conditions as no token or duplicated token, respectively. Clearly, the no token condition exists when the network is turned on initially.

To explain how the no token or duplicated token conditions are to be handled in ILLINET, let us discuss the error conditions that can occur in ILLINET. As described above, all RA's monitor the data stream on the ring as it passes by their 16-bit front-end windows. Data streams arriving at the optical receiver in a RA is not relayed to the optical transmitter unless this data stream represents the access token or when it is preceded by a flag and the flag is detected by the RA. At the end of the data packet, the last remaining 16-bit of data in the front-end window are delivered to the optical transmitter for transmission only when the occupied token marking the end of the data packet is detected. Hence, any data stream with no leading flag and occupied token is blocked by the front-end of some RA. A data stream with a leading flag but no occupied token is truncated by 16-bits after passing through each RA until the data packet disappears. A data stream containing occupied token but no leading flag becomes a control token instead. Thus, the continuous circulation of random or broken data packets left on the ring due to failures in the sending RA or intermittant noise is prevented. "Garage collection" in this case is not required.

In the sending RA, the data path between the optical receiver and transmitter is normally open during the transmission of a data packet. This data packet is removed from the ring when it returns. If for some reason the data path within the sending RA is closed when the data packet returns to the sending RA, it will be left circulating on the ring. We note that this error condition is a serious one. If the duplicate mark in the packet is not set and if the receiving process name matches the names of some active process on the ring, the input buffers in the RA serving these processes will eventually overflow since the data packet will be copied by these RA's each time it passes by their front end. In this case, a RA monitoring the network will see well-formatted data packets pass by even though the no token condition exists. Since the sending process name and packet sequence number are considered as parts of data and not monitored by the RA's, this type of no token condition can be detected either by the receiving hosts after the received data packets have been examined or by a RA after waiting for some access token for a period of time longer than the maximum access delay on the ring. If in a N-node ring with loops delay L the maximum packet length is T seconds, and each RA is allowed to transmit k times before freeing the token, the maximum access delay is roughly (N-1)(k)(L+T). (For example, in a 6 node, 1 km ring network, the length of this period is approximately 10 μsec. with k = 16.) Fortunately, we believe that this type of no token condition rarely occurs in ILLINET. When it does occur, it is handled as follows: when a RA observes data stream but no control token passes by its front-end window for a period of time longer than its estimated maximum access delay, it opens the data path between the optical receiver and transmitter. Thus, it removes the "garbage" from the ring. However, the no token condition persists.

The no token condition can be detected easily in the case when there is no data stream circulating on the ring. In this case, a RA can decide that there is no token on the ring after one maximum loop delay. (In our previous example, this time is roughly 7 μsec.) This type of no token condition is handled in the following manner. When a RA observes no data stream in the ring and there is no access token passing by its front end for a period longer than one maximum loop delay, it will enter a time-out period and continue to monitor the activities on the ring. If when its time-out expires and no token is observed, it will insert a token on the ring. By making the differences between the time-out periods of the different RA's equal to or longer than one loop delay, we are assured that once such a no token condition occurs, a token will be generated in a reasonably short time. Moreover, only one token will be generated in most cases.

The duplicate token detection scheme is designed for the general case when the exact loop delay is not known or may be variable. In this case, the duplicate tokens can be detected reliably at the host level. That there are more than one RA transmitting data packets at the same time can be detected by the sending host by examining the sending process name and packet sequence number in the data field of the packets arriving at the optical receiver of its serving RA. However, the need of the host intervention will undoubtedly significantly lower the network throughput. Alternately, we may require that the sending process name be placed in the first 16 bits of the data field. The sending RA can, therefore, determine whether the packet arriving at the optical receiver is the same one sent on the ring by itself. When the received data packet is found to be from another RA, the sending RA can conclude that duplicate token conditions exist. Again, by removing all data streams arriving at its receiver, a sending RA will delete all tokens from the ring.

5. <u>Hardware Structure</u>

The hardware structure of the RA's already implemented is described by the block diagram shown in Figure 5. To satisfy the different speed requirements of the different functional blocks of the RA at a minimum cost and complexity, it is implemented in ECL, STTL, and LSTTL. A RA consists of three PC boards, the front-end, memory module and retransmission control logic, and RA-to-host interface.

The front end contains the transmitting and receiving logics. Between the optical receiver and transmitter, there is a shift register which serves as a delay buffer. The RA may hold up the incoming data

ORIGINAL PAGE IS
OF POOR QUALITY

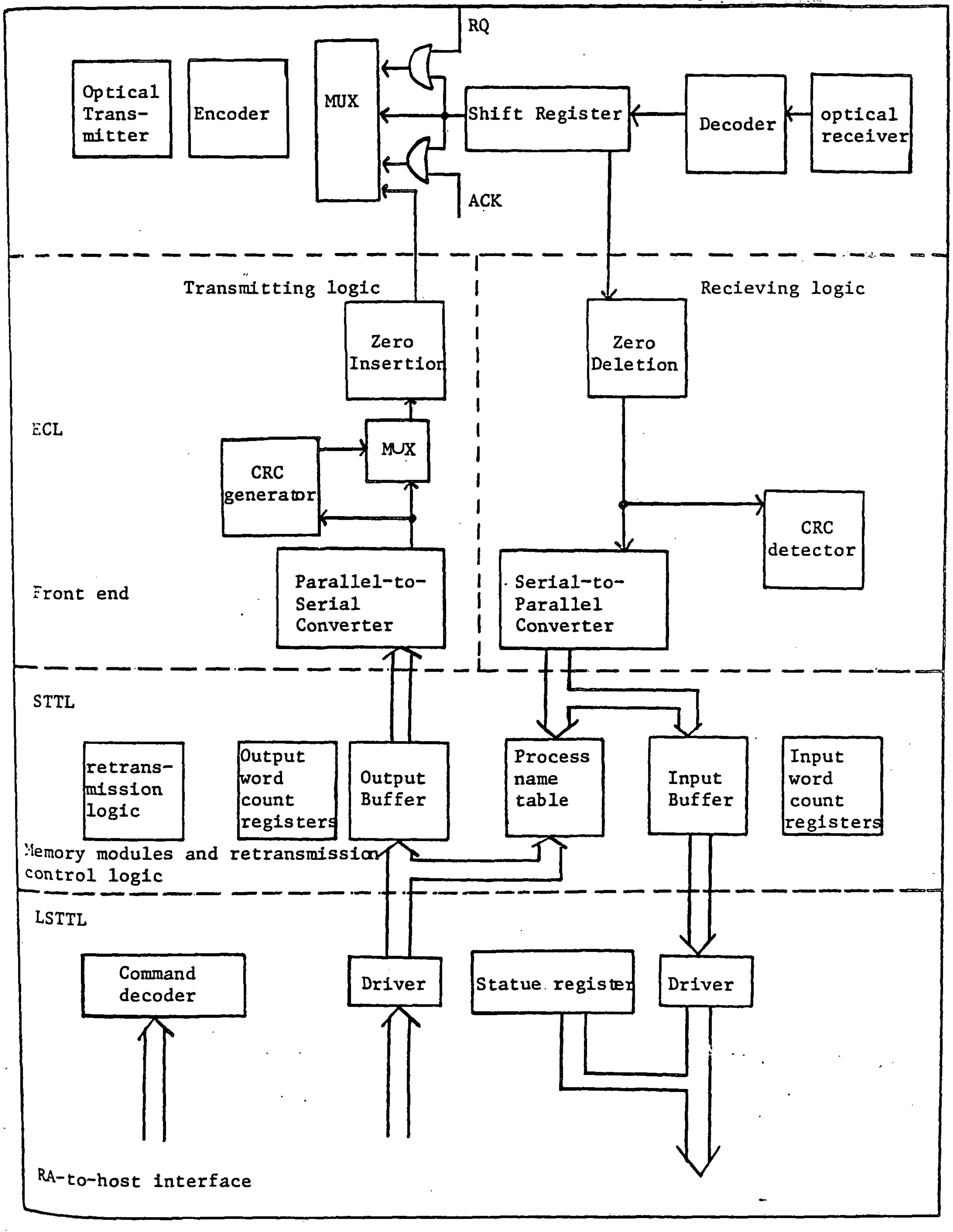


Figure 5

stream, scan and process the contents of the various fields as they appear in the shift register. Here, appropriate comment is generated and inserted in the ACK/RQ field of the data packet. Then the last 16 bits containing the ACK/RQ field and the token are shifted out to the transmitter. The major functions of the transmitting logic are parallel-to-serial data conversion, zero insertion CRC error code generation, and data packet formatting. The major functions of the receiving logic are zero deletion, serial-to-parallel conversion and CRC check. All these operations are carried out bit-serially and are implemented in ECL logic.

Within the memory modules and retransmission logic, the are input and output buffers, process name table, and retransmission control circuits. The buffer memory are segmented into 256 16-bit word pages. Sixteen pages are used as input buffers and two pages are used as output buffers. The input and output buffers are organized as independent modules, each is capable of supporting either read or write operation at 32 Mbits/sec. Upon detection of the flag, the input buffer write operation is initiated. If at the time data is being transferred from the input buffer to the host, this transfer operation is halted temporarily. The input buffer write operation will be terminated when the occupied token marking the end of the data packet is detected, when the receiving process name in the data packet does not match any names in the process name table, or when the duplicate mark is found to be set indicating that the date packet is already copied by the RA. Any temporarily halted memory transfer operation will then be resumed.

There are two output buffers in the output modules to allow the process of waiting for network access and data transfer from the host to be carried out concurrently. A 32 Kx1 memory module implemented with Intel 2147H3 is used to store receiving process names. (Currently, we are using only one chip containing 4 K in each RA.) The 15-bit receiving process name is used to address this RAM table. An output bit from the table being 1 indicate a match of the receiving process name with some local process name in the table. Thus, the process of checking receiving process name match can be carried out in 55 nsec. The table can be dynamically updated by the host within 500 nsec. All buffer memory operations, receiving process name checking and updating, and retransmission control are carried out at word level and are implemented in STTL logic.

Finally, the RA-to-host interface contains the command decoder, RA status registers and interfaces to and from buffer memory modules. These circuits allow the RA to appear to the host as a peripheral device and can be easily linked to the host via a DMA interface. This portion of the RA is implemented in the TTL logic.

Acknowledgment

The authors wish to thank Cyrus Weise, Kurt Horton, Izumi Suwa for suggestions and help in the design and implementation of ILLINET.

References

[1] Metcalfe, R.M. and D.R. Boggs, Ethernet: distributed packet switching for local computer network, CACM 19, 7, July 1976, 395-404.

[2] Frazer, A.G., Spider--an experimental data communication system, Proceedings International Communications Conference, 21F-1-10, CACM.

[3] Pogram, K.T. and D.P. Reed, The MIT laboratory for computer science network, Local Area Networking NBS Special Publication 500-31, April 1978, 22-23.

[4] Weber, H., D. Baum and R. Popescu-Zeltin, ESA--an evolutionary system architecture for a distributed data base management system, Proceedings of Berkeley Conference on Distributed Processing, 1979.

[5] Farber, D.J., A ring network, Datamation, February 1975, 44-46.

[6] Burton, H.O. and D.O. Sullivan, Error and error control, Proceedings of the IEEE 60, 1972.

[7] Mockopetris, P., Design consideration and implementation of ARPA LNI nametable, University of California, Dept. of Information and Computer Science, Technical Report 92, Irvine, CA, April 1978.

APPENDIX G

Shortest Job Next Scheduler

Shortest Job Next Scheduler

The Path Pascal scheduler implementation presented here is based on the FIFO scheduler presented in the Path Pascal User's Manual[Campbell & Kolstad 80c]. Each scheduler call is associated with an event descriptor. The event descriptor is entered into a queue of suspended jobs ordered by job estimate. The event descriptor is defined as follows:

```
type event_dscr = object
   path signal ; wait end;
   entry procedure; signal; begin end;
   entry procedure; wait; begin end;
end;   (* event_dscr *)
```

The event descriptor is part of a job queue element:

```
bufptr = ^buffer;
buffer = record
   job    : event_dscr;   (* an object for block/ unblock *)
   jobest : integer;      (* job time estimate           *)
   next   : bufptr;
end;
```

These are elements in a queue of suspended processes ordered by job time estimates:

```
SJNQ = object        (* Shortest Job Next Queue *)
   path 1: (enter ; leave) end;
   var head : bufptr;
   entry procedure enter (m: buffer);
         begin
            (* insert a job into the queue by jobest *)
         end;
   entry function leave : bufptr;
         begin
            (* return the head of queue.. lowest jobest *)
         end;
   init; head := nil end;
end;  (* SJNQ *)
```

These components are used to build the scheduler:

```
scheduler : object
   path suspend , resume end;
   var SQ: SJNQ;     (* the shortest job queue *)

   entry procedure suspend (jobtime: integer);
                 (* Block an incoming resource scheduling call until it
                 becomes the job with the lowest time estimate. *)
         var jptr: bufptr;
```

```
        begin
           new(jptr);
           jptr^. jobest := jobtime;
           SQ. enter (jptr);
           jptr^. job^. wait;     (* blocks until awakened *)
        end;

     entry procedure resume;
              (* Take the shortest job from the queue and release
              it so that it may execute its resource access. *)
        var jptr : bufptr;
        begin
           jptr := SQ. leave;     (* dequeues shortest *)
           jptr^. job^. signal;   (* unblock *)
           dispose (jptr);
        end;
  end;     (* scheduler *)
```

The resource is encapsulated in a separate object:

```
  resource : object
     path use_scheduled_resource end;
        ....
     entry procedure use_scheduled_resource;
        ...
     end;     (* use_scheduled_resource *)
  end;     (* resource *)
```

All processes that use this resource must execute the sequence:

```
  scheduler. suspend (my_job_estimate);
  resource. use_scheduled_resource;
  scheduler. resume;
```

Notice that if any user process does not follow this sequence of calls, serious problems result. If a process fails to call resume, the scheduler will deadlock. If a process does not call suspend, the scheduling is subverted and the result will probably be a corrupted resource. Simply implementing a sequential path in the scheduler:

path suspend ; resume **end**;

will not correct this problem. Although such a path will force every call to *resume* to be preceded by a call to *suspend*, there is no way to specify that a particular process must call both *suspend* and then *resume*. Object Path Expressions could be supplemented by Process Path Expressions[Campbell 76], Access Right Expressions [Kieburtz & Silberschatz 83] or capabilities [McKendry & Campbell 80c, McKendry & Campbell 82] to enforce such calling disciplines.

APPENDIX H

A Scheduling Mechanism

A Scheduling Mechanism

The following program uses a Path Pascal-like notation to present a specification for a shortest job next scheduler implemented using such a scheduling mechanism:

```
resource : object
   path 1 by job_est : (use_resource) end;
        ...
   entry procedure use_resource (job_est : integer);
        ...
   end;   (* use_resource *)
end;   (* resource object *)
```

A process calls this resource in this manner:

```
resource. use_resource (my_job_est);
```

The resource could be scheduled by substituting queueing on the least *job_est* for default FIFO queueing in the operation prologue and epilogue:

```
prologue:
   P (sem1 , job_est);      (* semaphore 1 *)
body:
   ...
epilogue:
   V (sem1 , job_est);
```

The P and V operations could be implemented as follows:

```
P (s, x) =
   s. value := s. value -1;
   if s. value < 0
   then begin
      (* queue in s.list by x *)
        block;
   end;
```

```
V (s, x) =
   s. value := s. value + 1;
   if s. value <= 0
   then begin
        P := (* call with least x from s. list *)
        unblock P;
   end;
```

Scheduling is appropriate only where Path Expressions place restrictions on the number of processes that may be executing. For example, it would be inappropriate to add scheduling to these paths:

```
path x , y end;
path [x] end;
```

Scheduling is useful where processes may be suspended:

```
path n by schedule_exp : (x) end;
path x ; by schedule_exp y end;
```

Of course, if the scheduling criterion is not specified, some default (such as FIFO) may be used.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1035	2.	3. Recipient's Accession No.
4. Title and Subtitle ILLINET--A 32 Mbits/sec. Local Area Network*			5. Report Date October 1980
			6.
7. Author(s) W.Y. Cheng, S. Ray, R. Kolstad, J. Luhukay, R. Campbell, J.W-S. Liu			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois 222 Digital Computer Lab Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NSF MCS 79-06945
12. Sponsoring Organization Name and Address National Science Foundation Washington, D.C.			13. Type of Report & Period Covered
			14.

15. Supplementary Notes

16. Abstracts

ILLINET is a fiber-optical ring network designed to provide wide band linkages between host computers for the purpose of facilitating file transfers at speeds near those of fast I/O devices in the hosts. Its structure is similar to the Distributed Computing System. ILLINET will eventually connect several PDP-11's, a PRIME computer and a network of microcomputers. These computers are used in a variety of real-time and batch processing applications. Currently they are already interconnected via 9600 band lines in a star configuration to provide access to simple terminals. This paper describes the network architecture, control structure, and hardware configuration of ILLINET.

17. Key Words and Document Analysis. 17a. Descriptors

Local-Area network
Ring network

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 28
	20. Security Class (This Page UNCLASSIFIED	22. Price