TOWARDS FAULT TOLERANT PROCESS CONTROL SOFTWARE[*]

Fred B. Schneider
Richard D. Schlichting

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

The construction of fault tolerant software for a multiprocessor consisting of N processors, each of which has access to its own local memory and to an N-port shared memory is considered. Software fault tolerance is achieved by structuring a program as a collection of cyclic processes with well defined communications channels and by using a simple protocol involving checkpoints.

Introduction

The use of computing systems to control complex devices or physical processes is becoming increasingly important. LSI and VLSI technology have made such systems inexpensive and small. Furthermore, such computing systems can respond very quickly to the events they monitor -- orders of magnitude faster than a human controller. This permits the control of complex, time-critical, physical processes, such an nuclear fission and air traffic.

Software intended to monitor and control such physical processes is called _process control software_. _Sensors_ are used to determine the state of the physical process by reporting values of key parameters and/or by detecting _events_ -- state changes in the physical process. _Actuators_ are used to control the physical process.

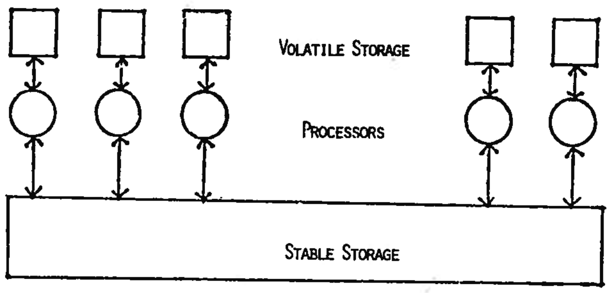
In addition to implementing a specified relation between input and output, process control software must also satisfy real-time response constraints or the ability to control the physical process may be lost. For example, altering the course of an airplane to avert a mid-air collision must be done promptly if disaster is to be avoided. For similar reasons, process control software must be fault tolerant.

Duplication of hardware is not sufficient to realize fault tolerance. Software protocols must also be devised so that after a failure, the system can be reconfigured and restarted in an orderly fashion. If a program is structured as a collection of cyclic processes with well-defined inter-process communications channels -- and most process control programs are -- then simple protocols can be used to achieve fault tolerance. The design of such a protocol is the subject of this paper.

[*]This work is supported by NSF Grant MCS-76-22360.

An Abstract Machine

A convenient abstract model of a computing system consists of N processors, each of which has access to its own local memory and a single N-port shared memory, as shown below:



Implementation of this abstraction will be discussed later. Here, we merely enumerate the pertinent properties of the components.

First, we assume that the local memory connected to each processor is _volatile_. That is, following a processor failure, the contents of its local memory are lost. On the other hand, the shared memory is assumed to be implemented as _stable storage_ -- it never fails and its contents are unaffected by any failures in the computing system. Although it is impossible to implement such a storage device, storage systems that approximate stable storage to any desired degree can be built.

Secondly, we assume the system interacts with any physical processes it monitors or controls solely through sensors and actuators. These are accessible to all processors. It is convenient to view these I/O devices as being memory mapped. Since it makes no sense to write to a sensor, memory locations that correspond to sensors are read-only with respect to processors. Even so, the value of such a memory location can change -- it does so as the quantity being measured by the sensor changes. Similarly, an actuator can be viewed as a write-only memory location. Clearly, the effect of writing a given value to such a location is dependent on the nature of the associated actuator. However, we require that actuators satisfy two properties:

A1: It is a change of value that initiates action by the actuator.

Hence, writing the same value successively to a given actuator memory location will cause no change to the state or operation of the corresponding actuator. Secondly,

A2: The length of time that an actuator remains at a given value does not affect the physical process.

Actuators that do not exhibit these properties make designing fault tolerant programs a hopelessly difficult task, since it is impossible to predict how long an actuator will remain at a given value if failures can occur. So an actuator found on a rocket that fires the thruster as long as its associated location has value 1 does not satisfy A2, since the length of time a thruster is fired presumably affects the velocity of the rocket.

Failures in a computing system can be viewed as events that must be detected and acted upon. As such, we assume the existence of a set of sensors that reveal the status of the remainder of the computing system. Thus, all failures of interest are detected. These sensors are merely a convenient way of viewing the fault-detection schemes employed in the computing system. We also assume that after a processor failure is detected, the offending processor is stopped*.

The literature offers many techniques for handling hardware failures -- both transient and permanent -- in a manner that is transparent to an executing program. For example, if a failure is detected while accessing memory, hardware might automatically retry the operation. Often, disk storage devices contain extra space intended specifically for use if a portion of the disk surface becomes unusable for some reason. The view taken above -- that failures are detected by sensors -- does not preclude use of such techniques. Rather, our model assumes that it is only when the hardware is unable to cope with a failure that the software is notified.

The Protocol

Much process control software is composed of communicating cyclic processes. This stems from the nature of process control applications, repeatedly: read from some sensors, do a computation, and write to some actuators. In the sequel, then, all processes are assumed to be of the form:

```
p_i: process
        <initialization>;
        do true → LB od
```

where LB is an arbitrary program called the loop body. (Where convenient, we use the guarded command notation of Dijkstra [Di76].) We seek a protocol to restart such processes after failures.

*A failure in a local memory is treated as a failure in the corresponding processor.

The system state is characterized by a collection of variables. Each variable is owned by one process. A process can read or write variables it owns, but only read variables owned by other processes. Thus, interprocess communication is possible, although in a somewhat disciplined manner. Sensors and actuators are not considered variables; hence they are owned by no process.

Variables are classified as being either history variables or temporary variables. A variable v is a history variable if:

H1: v is read by a process other than the one that owns it, or

H2: v is read in LB before it is written (i.e., it must retain its value across loop iterations).

Otherwise, v is a temporary variable. It is fairly simple to determine at compile time how to classify a variable by using standard flow analysis techniques [H77].

After the failure of a processor P, a reconfiguration rule is used to assign processes that were running on P to working processors, and a restart protocol is used to ensure that at no time do history variables or actuator memory locations have values they could not have had if the failure had not occurred. Thus, processor failures are made transparent to processes, except for possibly increased execution times due to the time required to reconfigure and restart the system. Determining a suitable reconfiguration rule is usually application dependent, so discussion of such matters is deferred until later. A restart protocol for cyclic processes is developed below.

After failure of a processor P, some state information concerning processes running on P might be lost because local memory is volatile. This suggests that processes be restarted at some well-defined point in their execution -- a point where sufficient information about the process state is available. Such points have been called checkpoints [De76]. The beginning of LB can be used as a restart point, provided P1 - P3, below, are satisfied:

P1: The history variables of a process are stored in the shared memory.

The contents of local memory will never be required at the beginning of a loop iteration if only temporary variables are stored there. Therefore, loss of the contents of local memory prior to an execution of LB is not catastrophic -- the relevant information is available in the shared memory.

Restarting a partial execution of LB should not change the number of modifications an actuator memory location undergoes. Thus,

P2: No actuator is written to more than once per loop iteration.

This is sufficient because the length of time an actuator remains at a particular value is irrelevant; only the number of transitions made by the actuator can have an effect on the physical process being controlled*.

Lastly, to ensure that LB can be restarted without adversely affecting its subsequent execution, the values of all history variables must reflect a complete execution of the loop, not a partial execution that was abnormally terminated. Furthermore, history variables can be updated only after changes have been made to actuator memory locations. Then, restarting LB will not cause actuator memory locations to undergo additional transitions. Hence, we require

P3: History variables modified in LB are changed:
- (a) once per loop iteration,
- (b) all at once as an atomic action,
- (c) after all actuator memory locations.

Below, we describe the implementation of such an atomic write operation based on a variation of the intentions list scheme of [LS78]. It ensures that either all of the values are written or none are, irrespective of the occurrence or timing of processor failures.

Implementing Atomic Write Operations

Let the history variables of process p_{id} be $v_1, v_2, \ldots, v_n$, and the other variables referenced by p_{id} be $x_1, x_2, \ldots, x_r$. Then, a variable named $switch_{id}$ and the 2-element arrays $v_1[0:1], v_2[0:1], \ldots, v_n[0:1]$ are allocated in shared memory. The loop body of process p_{id} is changed as follows:

```
LB: for i := 1 to n do
        $v_i[(switch_{id}+1)$ mod $2] := v_i[switch_{id}]$
    end;
    LB';
    $switch_{id} := (switch_{id}+1)$ mod $2$
```

where LB' is obtained from LB by changing:

(1) "v_i" to "$v_i[(switch_{id}+1) \bmod 2]$".
(2) "x_i" to "$x_i[switch_p]$" for each variable x_i owned by process p, $p \neq p_{id}$.

Notice that changing the value of $switch_{id}$ causes assignments made to the history variables during this loop iteration to take effect. Presumably, storing in a single memory word is an atomic operation, and therefore the desired multiple-value atomic write results. The for-loop prefix ensures that variables unaltered during a particular execution of LB' retain their values after such an atomic write. If all history variables are always assigned new values in LB', this loop can be omitted.

Processes communicate by using shared variables. We assume these variables are partitioned into resources, where a resource is a collection of variables that are related by some consistency constraints. Accesses to variables in resources are controlled to ensure that one process does not change the variables of a resource while another process is reading them. Otherwise, a process could read a set of values that does not satisfy the consistency constraints -- values that correspond to a partially completed update operation on the resource. Semaphores [Di68] can be used to guarantee the necessary mutually exclusive access to the shared variables comprising a resource.

*That is, the number of times the value of the corresponding actuator memory location is changed.

In order to preserve the consistency of a resource, each resource R will have associated with it a semaphore s_R that is initialized to 1. Further, all references to variables in R will be bracketed by $P(s_R)$ and $V(s_R)$ such that the V-operation is done only if the shared variables comprising the resource are in a consistent state. A section of code so bracketed is known as a critical section.

The possibility of failures complicates things somewhat because a process might then be halted while executing in a critical section. Assume p has been halted while executing in the critical section protected by semaphore s_R. Then, no process (including p) will be able to enter a critical section protected by s_R since the last process that entered never performed a V operation. To circumvent this difficulty, we define a synchronization mechanism that allows a process to re-enter a critical section if it had already entered that critical section, but never done a V operation to exit. A restartable binary semaphore rs is defined in terms of a non-negative integer variable that is set to the name of the process currently in the critical section, or 0 if no such process exists. Discussion concerning implementations of restartable semaphores is deferred until the discussion on implementing the shared memory; the two are closely related.

If history variables are accessed by more than one process, then these variables will be members of resources, and references to these variables must be from within critical sections. Let $rs_1, rs_2, \ldots, rs_m$ be the restartable semaphores used to protect the resources containing variable(s) owned by process p_{id}. Then, the body of the loop comprising process p_{id} should be replaced by the Synchronizing Protocol shown below:

```
LB: for i := 1 to n do
        $v_i[(switch_{id}+1)$ mod $2] := v_i[switch_{id}]$
    end;
    LB';
    for i := 1 to m do
        $P_r(rs_i)$
    end;
    $switch_{id} := (switch_{id}+1)$ mod $2$;
    for i := m to 1 do
        $V_r(rs_i)$
    end
```

where LB' is obtained from LB by changing:

(1) "v_i" to "v_i[(switch$_{id}$+1) mod 2]",
(2) "x_i" to "x_i[switch$_p$]" for each variable "x_i" owned by a process p, $p \neq p_{id}$,
(3) and bracketing all references in LB' to shared variables with P_r and V_r operations on the semaphore associated with the resource.

Note that the order in which P_r and V_r operations are performed is critical if deadlock is to be avoided.

Caveats

There are some assumptions implicit in the above protocol. First, whether or not a process control program satisfies its response time constraints is contingent upon the existence of sufficient processing power. Clearly, if enough processors fail, there may no longer be enough processor cycles available to meet these constraints. Here we have assumed that this will not occur; in reality it might.

Secondly, suppose some process p executes its loop every t seconds. Thus, due to P2 and P3 p will change its history variables and actuator memory locations at most every t seconds. Let wc be an upper bound on the time it takes to execute the part of a loop iteration after LB'; that is, the time it takes to change the value of "switch" and do the necessary P_r and V_r operations. Similarly, let rcr bound the time it takes to invoke the reconfiguration rule and then transfer p to a working processor after a failure. Then, the time T between actuator state changes is bounded by:*

$$rcr \leq T \leq (N-1)(t+rcr-wc)$$

where N is the number of processors in the system. Not only does this have implications with respect to satisfying response time constraints, but it means that assumptions about program execution speed should not be made within a program. For example, time should be read from a clock, not computed based on the number of iterations a loop is known to have completed and the expected execution time of the loop body.

Implementing Stable Storage

We now turn attention to implementing the shared memory, stable storage abstraction. Three representative implementations are considered. The first uses a single, highly reliable random access memory. In contrast, the second and third do not require any special type of storage device, but instead employ replication of data on independent storage devices to implement a highly reliable storage system. The second approach requires a reliable broadcast facility, while the third approach requires only a reliable interprocessor communications facility. All three approaches implement only approximations of the stable storage abstraction. It is not possible to implement true stable storage using a finite amount of hardware.

*We have pessimistically assumed that p has run on a succession of processors, each of which has failed.

Since complete fault tolerance is impossible, one usually attempts to construct systems that are sufficiently fault tolerant for the application at hand. Once such design goals are known, the system designer can select a cost effective design from among various system organizations. Typically, "distributed" system organizations are easier to make fault tolerant than "centralized" ones, because in a distributed system there are no components on which the operation of the entire system is dependent. However, distributed systems are usually more difficult to design and program than centralized ones, due to the additional communication and synchronization required when no single entity has a complete and accurate view of the system state. Both centralized and distributed organizations are considered in what follows.

The Centralized Approach: Using a Single Random Access Memory

Hardware implementations of stable storage approximations exist. Such a storage device is usually constructed by using a non-volatile memory technology and storing enough redundant information with each memory word so that error correcting codes can be used to reconstruct any information that is lost due to hardware failures. The result is a fully centralized storage system. Consequently, among other things the physical destruction of the storage device would result in the loss of its contents. In addition, such storage devices are usually more expensive and slower than volatile storage devices.

In order for our protocol to work with such a stable storage approximation, an implementation of restartable binary semaphores is required. This is simplified considerably given an instruction that allows interlocked access to memory. On the IBM System 370 architecture [IBM] the Compare-and-Swap (CS) instruction is provided for this purpose; it is used below. In other architectures, similar instructions have been defined. For example, on the DEC VAX11 machines INSQHI, INSQTI, REMQHI can be used [DEC]. Note, however, that not all memory interlock instructions are powerful enough to implement restartable binary semaphores when failures can occur. For instance, we have been unable to devise an implementation that uses the Test-and-Set instruction, even though this instruction can be used to construct binary semaphores.

The effect of executing a Compare-and-Swap instruction is as follows:

```
CS(t,s,n): atomically
               if  t = s  →  s := n
               []  t ≠ s  →  t := s
               fi
           end
```

Then, for each restartable semaphore rs, one word of storage is allocated in the shared memory. Associated with each process p_i is a unique integer name n_i, such that $1 \le n_i \le 2^{32}-1$ (assuming 32 bit words). Also, p_i has a variable, t_i, in local memory, that is accessed only by that process. Then, implementations for $P_r(rs)$ and $V_r(rs)$ for process p_i are:

```
P_r(s):  t_i := 0; CS(t_i,s,n_i);
         do t_i ≠ 0 ∧ t_i ≠ n_i →
              t_i := 0; CS(t_i,s,n_i;)
         od

V_r(s):  s := 0.
```

<u>The Distributed Approach</u>: <u>Replication of Data</u>

Stable storage approximations can also be realized by replicating data in independent volatile memories. To accomplish this, a copy of each item to be saved in stable storage is kept in every processor's local memory. Then, in order to support the "shared memory" abstraction, a protocol is used to keep these copies identical. Numerous protocols have been developed for this; for example, most solutions to the multiple-copy consistency problem for fully and partially replicated distributed database systems will suffice [B80]. Below, two additional protocols are described. They exploit the fact that only the owner of a data item can update it. Consequently, our protocols are simpler than the usual distributed database concurrency control schemes where such assumptions cannot be made.

The first approach is based on a reliable broadcast facility. A bus or ring structured communications network [F73] can support this mode of communications. Or, by use of a reliable broadcast protocol like the one described in [SS80a], a reliable point-to-point computer communications network can support reliable broadcasts. The second approach uses a reliable point-to-point computer communications network, but does not involve broadcasts. Techniques for implementing reliable point-to-point computer communications networks are not described here. Such matters have received extensive treatment elsewhere [D79].

<u>Broadcast-Based Approach</u>. Assume an interprocessor communications facility with the following properties:

CF1: A processor can perform a <u>broadcast</u>, which is received by all functioning processors or no functioning processor, regardless of the timing or occurrence of processor failures.

CF2: Messages broadcast are received in the same order by all processors.

Property CF2 follows from use of a bus or ring for communications, or by including timestamps on messages. Presumably, CF1 is a consequence of the nature of the communications network and its protocols.

Processes read variables in stable storage by fetching the value of the copy in local memory. Writes to stable storage are accomplished by broadcasting an "update y to z" message, which notifies all processors to change the value of variable y to z.

A distributed implementation of restartable binary semaphores can be constructed as follows. For each semaphore rs, a queue $q_P(rs)$ is allocated in the local memory at each processor P. This queue need only be as long as the number of processors that perform $P_r(rs)$ operations. Process p_{id} running on processor P performs a $V_r(rs)$ operation by executing the following:

```
V_r(rs): Broadcast: "V_r(rs) by p_id".
```

To perform a $P_r(rs)$ operation the following is executed by p_{id}:

```
P_r(rs):
    if "P_r(rs) by p_id" not in q_P(rs)
       then Broadcast: "P_r(rs) by p_id";
    delay until: first element of
       q_P(rs) = "P_r(rs) by p_id";
```

In addition, associated with each processor P is a <u>stable storage manager process</u>:

```
ssm_P:  do true →
             receive message m;
             if m = "P_r(rs) by p_x" →
                   put m at end of q_P(rs);
             □ m = "V_r(rs) by p_x" →
                   delete all "P_r(rs) by p_x"
                      messages from q_P(rs);
             □ m = "update y to z" →
                   y := z;
             fi
        od
```

All "$P_r(rs)$ by p_x" messages are deleted from $q_P(rs)$ when a "$V_r(rs)$ by p_x" message is received because multiple copies of the message might have been broadcast. This can happen if a failure causes p_x to be moved to another processor after performing $P_r(rs)$. Note also that $q_P(rs)$ must be protected from concurrent access.

This implementation of restartable binary semaphores is similar to the distributed semaphores described in [S79]. The interested reader is referred there to see how the correctness of our implementation could be verified.

As shown above, when an "update y to z" message is received by the stable storage manager process, it updates local memory. CF1 and CF2 ensure that all copies of a variable in the various processor's local memories appear to be identical, provided the Synchronizing Protocol given above is followed, where writes to stable storage are replaced by broadcasts of the appropriate update message.

The effects of write operations to history variables owned by p_{id} do not actually become visible to other processes until the value of "$switch_{id}$" is altered. This allows the broadcast of update messages to be delayed until the end of LB'. Thus, p_{id} can broadcast the values of all its history variables in a few broadcasts immediately before broadcasting "update $switch_{id}$ to ...", instead of broadcasting an update message every time a history variable is changed in LB'. Depending on the logic of LB', this may or may not be a significant optimization.

Approach Without Broadcasts. In order to update a variable in stable storage without using a broadcast facility, a message must be explicitly sent to every processor. Since the processor performing an update might fail in the middle of this, a variant of the two-phase commit protocol [G78] is used to ensure that either all processors, or no processors change their local storage.

As before, the value of a variable in stable storage is obtained by reading the copy of that variable in local storage. Writes to history variables involve two steps. First, an update message is sent to each processor that is known to be functioning. Then, when all of the history variables for a given loop iteration have been updated, the value of "switch" is changed. To change the value of "$switch_{id}$", process p_{id} executes the Change "switch" Protocol below, where in steps (1) and (3) messages are sent to processors in the same order as the order of processors to which p_{id} would be transferred in the event of failure:

(1) To all processors send: "Prepare to change $switch_{id}$".

(2) Wait until a "P prepared" or "P failed" message is received on behalf of all processors P.

(3) To all processors send: "Change $switch_{id}$ to x", where x = $(switch_{id}+1)$ mod 2.

Thus, this sequence of steps replaces the assignment:

$$switch_{id} := (switch_{id}+1) \textbf{ mod } 2$$

in the Synchronizing Protocol. Note that in step (2) some facility for detecting processor failures and generating the necessary messages is assumed. For example, functioning processors could send "P failed" messages on behalf of those processors known to be no longer functioning. Also, "all processors" in the above protocol includes the one executing p_{id}.

Each processor P is assumed to have a stable storage manager process for every process p_{id} that owns variables in stable storage (including processes presently running on P). Let rs_1, rs_2, ..., rs_n be the restartable binary semaphores guarding resources that contain variables owned by p_{id}. If it is assumed that messages are always routed to the correct stable storage manager based on the originating process, then the stable storage manager process that runs on processor P for p_{id} is:

```
receive message m;
do m = "update y to z" → y := z
 □ m = "Prepare to change switch_id" →
     for i := 1 to n do; P(rs_i); end;
     reply: "P prepared";
     receive message m;
     if m = "Change switch_id to x" →
         switch_id := x;
         for i := 1 to n do; V(rs_i); end;
         receive message m;
      □ m ≠ "Change switch_id to x" →
         for i := 1 to n do; V(rs_i); end
     fi
od
```

Notice that regular binary semaphores can be used, because the processor to which a process is moved after a failure will have its own semaphores. The necessary system-wide synchronization achieved previously by using distributed restartable semaphores is realized here by the two-phase commit protocol.

If a processor failure occurs while a process is in the midst of executing step (3) of the Change "switch" Protocol, the value of "switch" might not be changed at all processors. This is clearly undesirable, since the values of all copies of each variable must be kept identical for the shared memory abstraction to be realized. Therefore, whenever a process p_{id} is restarted on another processor (after being moved due to a failure) the following protocol is executed:

(1) To all processors send: "Prepare to change $switch_{id}$".

(2) Wait until a message with text: "P prepared" or "P failed" is received on behalf of all processors P.

(3) To all processors send: "Change $switch_{id}$ to x", where x = $switch_{id}$

As before, in steps (1) and (3) the order in which messages are sent to processors is the same as the order of processors to which p_{id} would be transferred in the event of a failure. This is because, otherwise, the restart protocol could allow another process to observe p_{id}'s history variables regress in time.

Restarting Repaired Processors

After a failed processor has been repaired, it must be re-integrated into the system. In the centralized implementation of the stable storage abstraction, this is quite simple. Some processes are selected and moved to the repaired processor at the beginning of their respective loop iterations. Things are slightly more complex in the distributed implementations of stable storage, because the repaired processor's local memory must be restored, as well. This can be done in two ways. The first involves a protocol that allows a processor to request from other processors the current contents of stable storage. However, there is some subtlety in the design of such a protocol. In particular, while memory downloading is taking place, other processes might be updating history variables. So the protocol used must avoid multiple-copy consistency problems. A second approach to reinitializing a processor's local memory, which can take longer, is for a newly repaired processor to respond to update messages as they are received. After the processor has received two "Change switch ..." messages from each process that owns variables in stable storage, restoration of local storage is complete and processes can be moved to the repaired processor.

Optimizations for Practical Special Cases

There is some interaction between the assignment of processes to processors, the reconfiguration rule, and implementing a stable storage approximation. The various schemes outlined above will work in conjunction with any reconfiguration rule. However, for a given reconfiguration rule, optimizations are usually possible.

For example, sensors and actuators need only be accessible (hence connected) to processors that might actually run processes that reference these devices. For a variety of reasons, it may not be desirable to connect every processor to all sensors and actuators. Thus, the reconfiguration rule is determined, in part, by such connections.

In many situations the system can be expected to experience only a small number of processor failures. Then, it is known *a priori* to which two or three processors a given process might be assigned. This, and knowledge of the interprocess communications topology of the program can be used to govern assignments of processes to processors so that instead of replicating all variables at all processors, a copy of a variable is maintained at a processor only if some process that accesses that variable could run there.

Conclusions

Constructing a fault tolerant computing system involves defining a reconfiguration rule, so that after a failure the computing load can be repartitioned over the remaining working processors, and devising a restart protocol, so that at no time do program variables or actuator memory locations have values they could not have had if the failure had not occurred. Thus, a failure is made transparent to other processes in the system, and can have no effect on the environment being controlled by the computing system.

For a program S, a restart protocol for S, rp(S), is a program that can receive control at any point during execution of S -- presumably after a failure -- and establish the postcondition of S. In [SS80b], three conditions that ensure S' can serve as a restart protocol for S are developed. They are:

(1) variables mentioned in the precondition of S' are always defined,

(2) the precondition of S' is universally invariant over execution of S, and

(3) the precondition of S' is universally invariant over execution of S'.

Condition (1) guarantees that the data required by S' for successful execution is available after a failure; (2) is necessary because a failure can occur at any point during execution of S; and condition (3) is similarly required because a failure can occur at any point during execution of S'.

In this paper, a restart protocol for cyclic programs was presented. It was obtained by applying the theory described above, in the following manner. Associated with every program loop is a *loop invariant*, an assertion that is true at the beginning and end of each execution of the loop body. This loop invariant can be made into a universal invariant by requiring that all changes to variables in the loop invariant -- history variables -- be performed in a single atomic action. Then, the loop body can serve as its own restart protocol since conditions (2) and (3) follow from the universal invariance of the loop invariant, and condition (1) follows if history variables are stored in stable storage.

Lastly, it is worthwhile to point out that in our solution, we first postulated the existence of a stable storage abstraction and then considered its implementation. Of course, "separation of concerns" is not a new technique in constructing software. Nevertheless, choice of a suitable abstraction simplified things considerably. Stable storage appears to be a very useful abstraction in the context of fault tolerant software. It is used in [G78] and [LS78] as well.

Acknowledgments

The problem addressed in this paper was first suggested to us by J. Kemp, W. Comfort, and M. Kushner. We would also like to thank G. Andrews, R.W. Conway, and G. Levin for their comments on an earlier draft of this paper.

References

[B80] Bernstein, P.A., N. Goodman, Fundamental Algorithms for Concurrency Control in Distributed Database Systems, CCA-80-05, Computer Corporation of America, Mass.

[DEC] Digital Equipment Corp., *VAX11 Architecture Handbook*, Digital Equipment Corp, Maynard, Mass, 1979.

[D79] Davies, D.W., et.al., *Computer Networks and Their Protocols*, John Wiley and Sons, NY, 1979.

[De76] Denning, P.J., Fault Tolerant Operating Systems, *Computing Surveys* 8:4, 359-389.

[Di68] Dijkstra, E.W., Cooperating Sequential Processes in *Programming Languages* F. Genuys (Ed.), Academic Press, NY, 1968.

[Di76] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

[F73] Farber, D., et.al., The Distributed Computing System, in *Proc. CompCon 73*, Feb. 1973, 31-34.

[G78] Gray, J., Notes on Data Base Operating Systems, IBM Research Report RJ 2188, (Feb. 1978).

[H77] Hecht, M., Flow Analysis of Computer Programs, Elsevier North-Holland, NY, 1977.

[IBM] IBM Corp., IBM System/370 Principles of Operation, GA22-7000-3.

[LS78] Lampson, B., H. Sturgis, Crash Recovery in a Distributed Data Storage System, submitted to CACM.

[S79] Schneider, F.B., Synchronization in Distributed Programs, to appear in TOPLAS.

[SS80a] Schneider, F.B., R.D. Schlichting, Fast Reliable Broadcasts, in preparation, Cornell University Dept. of Comp. Sci.

[SS80b] Schlichting, R.D., F.B. Schneider, Verification of Fault Tolerant Software, TR 80-446, Cornell University Dept. of Comp. Science, Nov. 1980.