RC 20114 (04/05/95)
Subject Area

IBM Research Report

Restart Services for Highly Available Systems

Nicholas S. Bowen, Christos A. Polyzois and Richard D. Regan

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Abstract

This paper proposes a design methodology for building highly available systems. In addition, we describe a set of operating system services that can be used to achieve this goal. The techniques described are intended for a parallel environment and can be generalized for any distributed system. We describe a methodology for providing basic services for high availability, specific services for restart and an implementation of these services.

Keywords: Parallel Systems, Availability, Distributed Systems, Membership services

1 Introduction

As the role of information technology becomes more central in businesses, the reliance on computers becomes more critical and the requirements become more demanding in many aspects. There is an insatiable demand for higher performance which generally implies more hardware components. Requirements also increase along the functionality dimension which usually gets translated to more and larger software components. At the same time, the availability expectations are also rising rapidly, since the effects of downtime become more apparent and have higher economical consequences for larger systems.

In some sense the two trends are conflicting. The presence of more and larger components increases the chances that something will go wrong and tends to decrease availability. One way to increase availability is to increase the reliability of every component. However, failures are inevitable. Thus, we have to address downtime, the other key factor of availability.

It is often the case that a failure can be overcome by restarting the system or reconfiguring it to bypass the faulty component (e.g., [12]). In other words, one can reassemble the various components, either in the original way or in a new one, to continue providing service. Automation of this process is important, since this eliminates the long human reaction time and the likelihood of a human error. Managing the restart of a system may require several complex steps, so it is an error-prone procedure.

In this paper, we examine a set of services intended to facilitate this task. Section 2 defines our system model and presents the requirements that such a set of services should satisfy. Section 3 describes a particular implementation of a set of restart services. Section 4 generalizes these services beyond a tightly coupled system to a loosely coupled cluster. Section 5 concludes the paper.

2 Model and Requirements

We present our model for a computing system and the requirements which it imposes on the services that provide recovery/restart.

In general, a computing system has multiple processors. We consider both tightly coupled systems (e.g., a shared memory multiprocessor) as well as loosely coupled systems (e.g., a cluster of workstations). Multiple processors allow for incremental growth and can provide higher cumulative processing capacity than a monolithic system. From an availability standpoint, multiple processors are desirable because they allow continuation of service even if one of the processors fails. If enough capacity is reserved during normal operation, the performance impact of a failed processor may be very limited.

Each processor runs an operating system. Running a separate instance of the operating system on each processor (e.g., in a cluster of workstations [1, 6]) provides a convenient firewall, since it avoids contamination or sympathy sickness for other processors if one processor fails. This is harder to achieve if the processors share operating system data structures, which is often the case with symmetric multiprocessors. However, there are multiprocessor systems that can isolate a faulty processor and let the remaining processors continue.

A software subsystem is viewed as just another component in the system. Given the increasing reliability of hardware and the increasing complexity and continuous evolution of software, it is likely that software subsystems will become the dominant factor that affects system reliability.

It is usually possible to deliver service even with software that is not perfect. Many undetected software problems are workload dependent and are triggered by certain sequences of non-deterministic events (e.g., by a particular process timing). If the software component is restarted, the problem may not appear again (or it may appear later). Logging and checkpointing can also be used to ensure forward progress is made despite restart [2].

Our goal is to provide a set of services that facilitate recovery and automate restart processing. Eliminating the need for human intervention allows faster restart, and, therefore, shorter down time. Furthermore, since restarting a system may require a complicated series of operations, it is an error-prone procedure, so there is an additional advantage in obviating the need for human intervention.

The first element required for recovery and restart processing is to prepare for failures. This is done by first defining the expected set of failures and then specifying the appropriate restart or reconfiguration actions to be taken upon occurrence of each failure. The actions associated with a failure implement a certain policy. For example, in case a node fails in a partitioned (shared nothing) database, it may be desirable to start a new partition on a spare processor. In a shared disk database, it may be desirable to spread the load of the failed processor across the remaining processors, assuming they have spare capacity. This specification of recovery and restart policies requires extensive support from systems management tools.

The resources that are subject to failure must be monitored [3, 10]. This can be done either by periodic monitoring or by having one subsystem report another as a suspect, if the latter has failed to provide the service it is supposed to. For example, a failure to get a response from a remote NFS server can be reported and used as an indication that something may be wrong on the server node.

When a failure is detected [11, 13], all surviving processors must agree what failure has occurred and how it should be handled. Agreement is critical in order to avoid conflicting actions that may arise if two different processors try to react to the same failure (or possibly different perceived failures).

A critical observation is that the ability to reach agreement between processors must be present despite all (expected) failures. In a tightly coupled system, shared redundant hardware can be used to ensure that all processors can exchange information; such an approach is described in Section 3.

The algorithm for deciding the next operating point may be run on a single processor, or it may be a distributed algorithm involving multiple processors. In the latter case, the processors must have the ability to communicate and synchronize. Once the desirable operating point has been determined, the system must perform appropriate actions to move to the new operating point. Multiple (possibly all) processors may need to take actions. Some of the actions may involve isolating the possible cause of failure. For example, a suspect processor may be powered off or rebooted. As a less drastic measure, the suspect processor may be cut off from the disks to ensure it will not interfere with subsequent processing.

Most recovery actions will involve restart processing. These actions must be coordinated in two dimensions. One dimension is within a processor. For example, it may be necessary to start a file system before starting a database that stores its data in that file system. Similarly, the database system may need to be up and running before starting a transaction monitor that needs the services of the database system. Thus, it is necessary to specify ahead of time the timing dependencies between the various subsystems.

A restart/recovery model is necessary to enable expressing the timing dependencies between subsystems. Ideally, such a model should be flexible enough to allow various patterns of synchronization and maximal parallelism across subsystems. On the other hand, the model should lend itself to simple and efficient implementation, since it will be used to enforce the timing dependencies at run time. Furthermore, a simple, restricted model could be checked statically for desirable properties like termination and deadlock-freedom.

The second dimension along which coordination must occur is across processors. For example, a partitioned database may run one instance on each of several processors, and it may be necessary to start all instances of the database before starting a transaction monitor. In this case, the synchronization is between entities that belong to the same subsystem but run on different processors. It is desirable to provide a generic facility that allows distributed subsystems to synchronize their instances [5].

3 Restart Algorithms

In this section we propose a set of algorithms to perform restart processing. The algorithms are proposed as a complement to basic membership services and rely on failure detection at both the process and processor level. Section 3.1 describes the restart services as viewed from an operational process, Section 3.2 describes the processing that occurs after a failure, and Section 3.3 describes the robustness built into the restart services algorithms.

3.1 Basic Restart Services

Services are provided to allow a process to indicate when restart services are required (Register) and indicate when they are no longer required (Deregister). The parameters used on the register service are shown in Table 1. *Name* and *Type* are used to logically identify the process from an external source. *Prestart program* allows the process to specify a special program that should be run before the process is actually restarted. This can be used in situations where some cleanup activities must take place before the process can be restarted. *Prestart data* allows data to be passed to this program. *Restart command* allows the process to specify a command that should be used to restart the process. The operating system can normally figure out how to restart the process so this parameter is only used in special situations. *Failure Type* enables a special case of affinity, that is, the process can only run on the processor on which it registers. Therefore, this allows a process to prevent the operating system from restarting it on another processor in the event of a processor failure.

Parameter	Purpose
Name	Used to identify the specific process
Type	Generic name to identify a class of processes
Prestart program	Names a program to run before the restart
Prestart data	Data to pass to prestart program
Restart Command	A system command used to do the restart
Failure Type	Indicates action to take for various failures

Table 1: Input parameters for register service

The outputs of the register service are highlighted in Table 2. *Restart Type* provides an indication as to whether the process was restarted by the operating system or is initially starting. *Restarts Enabled* indicate if the operating system is currently enabled and capable of performing restarts. *Prior Processor ID* provides an identification of the prior processor on which the process was executing.

Output	Purpose
Restart Type	Indicates if process was restarted
Restarts Enabled	Indicates if restarts are currently enabled
Prior Processor ID	Identifies the previous processor the process was on

Table 2: Output Parameters

The other services are summarized in Table 3. The *Ready* and *Wait* services are used to synchronize processes which have sequencing dependencies (these services are described in detail in Section 3.2).

Service	Purpose
Register	Indicate that restart services are requested
Deregister	Terminate request for restart services
Ready	Indicate process has completed restart process
Wait	Allows process to wait for dependent processes
Associate	Used in hot-standby system – disable restart of primary process

Table 3: Summary of Commands

3.2 Restart Techniques

This section describes the specific algorithms used to perform the restart processing. Figure 1 shows the major epochs in the restart processing. We describe this process using the letters indicated on the figure.

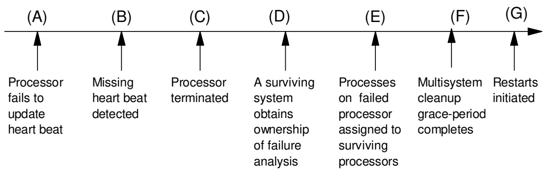

Figure 1: Major Epochs in restart

A We assume standard system-level heartbeat mechanisms exist. This first event indicates the first instance in which a processor fails to update its heartbeat.

B The missing heartbeat is now detected. The delay from A-B is a function of the sensitivity of the heartbeat technique to tolerate irregularities in the mechanisms. For example, a sharp transient in the steady state workload could cause a processor to miss a heartbeat. It is critical that this is specified correctly to quickly detect a real system failure while avoiding "predatory takeovers." We assume that the interval for detecting a missing heartbeat is a system parameter that has been correctly specified.

C Starting at time **B** actions must take place to ensure that the processor is actually terminated. This is required to tolerate the possible imperfect nature of some heartbeat mechanisms. For example, a processor could stop updating its heartbeat because of a loop in the kernel of the operating system. The missing heartbeat could be externally recognized and restart processing could occur after which some other event occurs that terminates the loop (e.g., an I/O error condition cleared, timeout logic in the operating system). The processor then comes back to life and there could be two copies of the processes that were executing on the apparently failed processor. This could cause database problems in a shared disk environment or network problems in a distributed system.

We assume the use of a technique that we refer to as "I/O Fencing" in which there is a special communications link to the failed processor's I/O system that allows I/O from the failed processor to be disabled.

D As soon as the processor is officially declared terminated in event **C** a signal is broadcast to all processors indicating this event. At this time each processor attempts to become the logical owner of the process to initiate the restarts. This could use any algorithm [4] for either tightly coupled or clustered systems. Our approach is to use a shared disk that uses logical locking on the disk. The processor that becomes the logical owner records its processor identifier on the shared disk.

E Once a processor becomes responsible for the restart process it begins a set of steps which lead to assigning the processes to execute on other processors and initiating their restarts.

1. Assign each process to an affinity-group of processes that must always remain together. These groups may consist of between 1 and N processes, where N is the total number of processes on the failed processor.
2. Assign each affinity-group to an available processor:
   (a) Honor any static affinities
   (b) Assign the affinity-group to a processor based on recent historical resource consumption (e.g., CPU, memory) of all processes in the affinity-group and resource availability on the active processor.

F The system provides a period of time for all members of groups to perform multisystem cleanup before the restarts are initiated. For example, suppose there was a multisystem lock manager that needed to clean up lock state before the processes are restarted. If a process requires this grace period, then it is specified when it joins a group. It is responsible for notifying the operating system when it has completed its processing.

G Event G is triggered when all members requiring the grace period have notified the system. The processor that was the logical owner at time **E** signals all processors that restarts should occur. The processes that need to be restarted are indicated on the shared disk containing the global state. The signal is used as a trigger to read the shared disk. Once the local processor determines which processes must be restarted the following steps occur.

1. If the process specified a pre-start program then execute the program. This program can perform many process dependent actions that must be done before the process is restarted.
2. For all processes in a given affinity-group, create a list specifying the order in which processes must be restarted. This is based on a parameter (level) which allows the specification of an integer used to determine this list. If multiple processes have the same level then they are logically started simultaneously.
3. Initiate the restart of all processes in the affinity-group using a simple pacing algorithm that specifies delays between process restarts.
4. Enable sequencing of processes within an affinity-group. The processes have READY and WAIT calls that allow them to wait for processes that were started earlier (i.e., that have a lower level).

The key aspects are illustrated in Figure 2 and described in detail in the following list:

1 All processes within an affinity-group are restarted with a pacing value to stagger the restart processing. The purpose of this parameter is to avoid a system overload (process initiation functions) or causing problems with individual processes (e.g., if many processes being restarted must all read the initialization file then one could avoid overloading the disk system by staggering the restarts). A pacing value of "0" essentially disables the pacing.

2 Processes $p(i)$ and $p(j)$ perform some initial restart processing but then desire to wait for the set of processes in the prior level to become ready (e.g., they may require services provided by these processes). $p(i)$ and $p(j)$ issue "WAIT" calls and their processing is suspended.

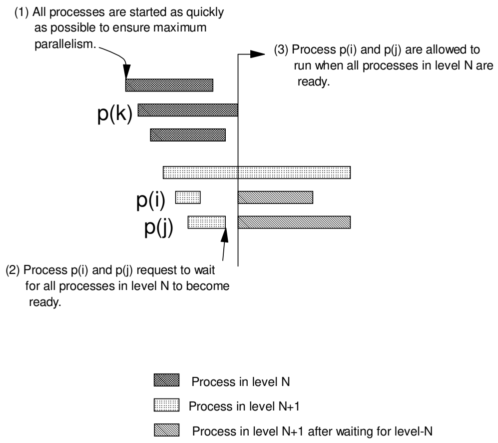

Figure 2: Parallelism in Restart Processing

3 When process $p(k)$ indicates it has completed restart processing (i.e., issued *Ready* command) then all processes in level N are ready which then means any processes in level $N+1$ that are currently waiting can resume processing.

3.3 Robustness in Algorithms

Since our objective is to provide operating system services to build highly available systems we must ensure that the services have additional fault tolerant features. In Section 3.3.1 we formulate a methodology for tolerating faults by precisely defining the way in which failures are exposed to users of the services. In Section 3.3.2 we provide examples of this methodology in the context of the suggested restart services.

3.3.1 Basic Fault Tolerance Model

Our design objective is a system which can recognize failures, remain operational while failures persist, and restore itself to normal operation when failed components are repaired. Figure 3 illustrates both the physical states (top line) and logical states (bottom line) that occur due to a failure. The physical states, indicated by numbers, are described as:

1. Normal operation.
2. A period of time from when the system has recognized the failure of one of its components and when it restores itself to a limited, yet safe, configuration.
3. A period of time from when the system establishes a safe configuration and reestablishes its original configuration.
4. Normal operation.

Failure masking is a key objective in this system. Therefore, we want to present a logical view to the user of the restart services that mask the physical state transitions just described. This is illustrated by the lower time line in Figure 3. That is, the users of restart services see a continuously available system with the caveat that some (logically) instantaneous change may occur as a result of an underlying failure. The system is designed such that processes that register before the logical failure point can be distinguished from those that registered after the failure. We describe these processes by set A (before failure) and set A' (after failure). This formalism is especially critical in a distributed or parallel system in which information (e.g., failure notification, repairs) is subject to propagation delays and thus may be recognized globally at different points in time.

Achieving this logical view mandates additional design considerations to handle events that occur when the underlying system is repairing a physical failure. Solutions include techniques such as queueing the requests or rejecting them and notifying users upon repair (this is essentially queueing in the operating system or queueing in the user).

3.3.2 Example

This section illustrates two specific examples of the basic fault tolerance model.

Processor Termination

When a processor fails there are two fundamental actions which must occur. First, the processes from the failed processor must be restarted on active processors. Second, the processor itself must be restarted. If these two activities can be performed in isolation from each other then the implementation is greatly simplified. However, if these activities are independent then one must be careful regarding processes that register after the processor is restarted (i.e., they should not appear as candidates for process restart).

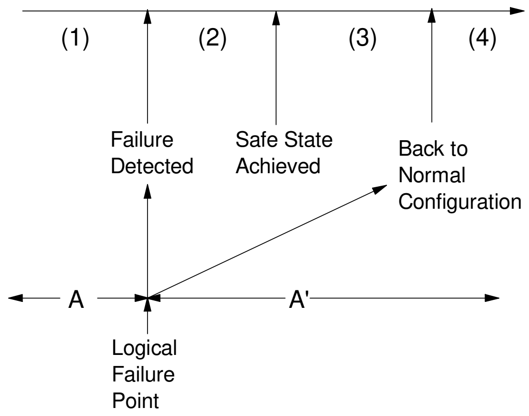

Figure 3: View of failures

Figure 4 is used to illustrate the application of this methodology to a processor failure scenario. As described in Section 3.2, a processor failure results in the processes being assigned to another processor. Using our design methodology we are able to simplify the overall design by examining all registered processes (recorded on the shared disk) and distinguishing those processes that must be restarted (i.e., those active before the failure) from those that should not be restarted (i.e., those that registered after the processor restarted).

Loss of access to the shared disk global state

As we stated in Section 2, the information required to determine the necessary reconfiguration must be accessible in case of failure. In our system, this information is kept on a shared disk, with independent access from every processor to the disk. To guard against failures, the disk can be mirrored and the hardware path to the disk can be duplexed.

As processes register and deregister, recovery actions change, and must be reflected on the disk. If a processor ever loses access to the shared disk containing the global state, the state as reflected on the disk may diverge from the actual state of the system. Consider the following scenario:

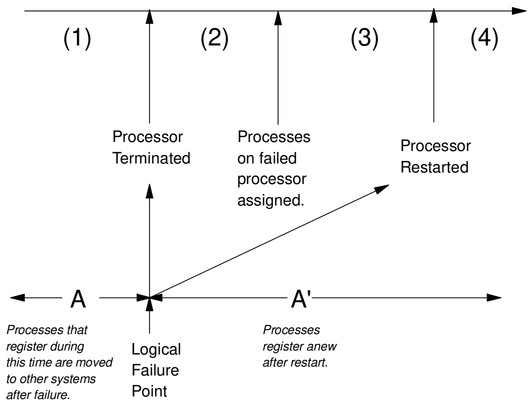

Figure 4: Realization of Processor Failure

1. A process deregisters itself, but the processor cannot access the shared disk to record the deregistration.

2. The processor on which the process is running terminates, and the process is (incorrectly) restarted on another processor.

The net result is an undesired instance of the process, which could cause data races and data corruption (especially if another instance of it has started somewhere else). Starting one process too many is generally more dangerous than starting one too few. A missing process will not provide service, but at least it will not compromise data integrity.

One could handle the loss of *physical* access to the shared disk global state by substituting *logical* access for it. The process registration/deregistration actions could be sent to other processors, and recorded on disk by them. For resiliency, the actions would have to be sent to many (all) other processors, and it would be necessary to arbitrate among those processors to ensure exactly-once semantics for the actions. Furthermore, appropriate algorithms would be necessary to ensure that the logical order of the actions on the processor of origin (the one that lost access to the disk) is preserved. For example, the relative order of two registration/deregistration events for different processes (or even the same process) should not be reversed by messages received out of order by other processors.

The approach outlined above would require a lot of effort and consume a significant amount of resources (broadcast for every registration/deregistration event, contention for the shared disk, etc.). The gain in service availability would be minimal, because the loss of access to the shared disk is a very rare event, given that it requires at least a double failure (single failures are masked by the redundant hardware used).

We have opted for a solution that brings the system to a safe state when a processor loses access to the shared disk global state, rather than provide maximal service. A safe state is chosen such that restart services are made logically unavailable on the affected processor (e.g., new registration requests are not honored). More precisely, this safe state is one in which it appears that restart services were never available on the affected processor; that is, processes are logically deregistered, which prevents undesirable restart scenarios like the one outlined above.

Figure 5 shows the state transitions for lost access to the global state. Again, we are able to distinguish processes that present themselves before (set A) and after (set A') the logical failure. The algorithm for deregistering processes in set A is an asynchronous message based algorithm, so we must be able to guarantee that processors that perform the deregistration remove only those processes in set A and do not touch processes in set A'.

3.4 Techniques for Recovery of Global State

This section describes the specific algorithms used to recover from a loss of access to the shared state. The key to the robustness of these algorithms is the use of sequence numbers to logically bind registered processes with a specific configuration instance. There are two sets of sequence numbers that are defined:

1. An array of sequence numbers PSeqNum(), which contains one sequence number per processor. Each sequence number PseqNum(P) represents the nth connection to the shared state by processor P. In the absence of access failures, this sequence number contains the value of one. In the event that access is lost and eventually regained, this sequence number is incremented.

2. An array of sequence numbers pSeqNum(), which contains one sequence number per registered process. Each sequence number pSeqNum(p) is a copy of PSeqNum(P) as

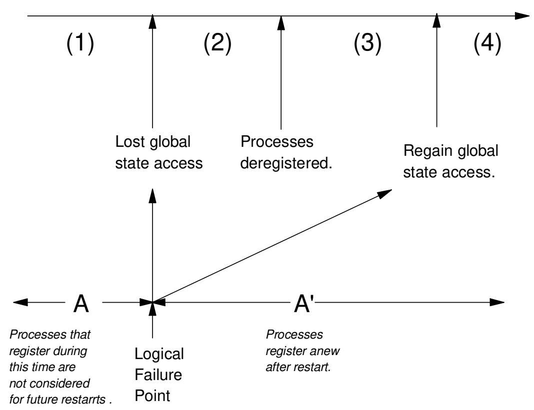

Figure 5: Realization of global state failure

it existed at the time process p registered on processor P. This represents the bind between a process and the particular configuration under which it registered.

Each processor tracks every other processor and maintains a logical (persistent) copy of the PseqNum array. In addition, each processor tracks which processors currently have access to the shared disk state and which do not.

Figure 6 shows the major epochs in recovering from a loss of access to the global shared state. We describe this process in terms of the letters indicated on the figure.

I Processor P1 makes its initial connection to the shared disk. PSeqNum(P1) is initialized to one (i.e., initial access). When process p registers on processor P1, its state data representation is tagged with P1's access sequence number. That is, pSeqNum(p) is set to PSeqNum(P1).

J An attempt to access the shared state fails. A "lost state" signal containing P1's logical processor identifier and PSeqNum(P1) is broadcast to all other processors to request that processes registered on processor P1 be deregistered. Any processor with access to

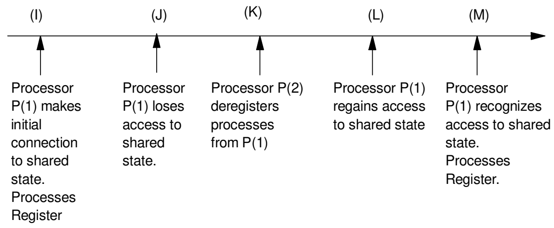

Figure 6: Major epochs in recovery of global state

the shared state is eligible to perform the deregistration process on behalf of processor P1.

K Processor P2, which still has access to the shared state, receives the signal from processor P1 and logically deregisters all P1 processes that are tagged with a sequence number that is less than or equal to the sequence number in the signal. Included in the processes that are deregistered are processes that originally registered on P1 and processes that have been assigned to P1 but have not yet been restarted.

Processor P2 also updates the state information to indicate that processor P1 no longer has access to the shared state. In addition, each affected process on processor P1 is notified that it has been logically deregistered.

Note that other processors may attempt to deregister P1's processes, but only P2 will find any processes to deregister.

L At some time in the future, the path to the shared state from processor P1 is repaired, and processor P1 regains access. Before reaccess is officially recognized, processor P1 attempts to deregister all processes that may still be registered with a previous

PSeqNum(P1) (this is the same processing that is performed in step **K**). This step is taken as a precautionary measure in case the "lost state" signals are lost.

M Processor P1 increments PSeqNum(P1) (i.e., its value is now 2) and updates the state data to indicate that it now has access to the shared state. Notification is broadcast to all "listening" processes on P1 so that they know they can successfully reregister.

The above algorithms ensure that any process that registers prior to a loss of access will be deregistered, and that any process that registers after reaccess will be treated as any other normally registered process (i.e., it will be restarted if its processor fails). These algorithms tolerate the following timing anomalies:

All "lost state" signals are lost.

When a processor regains access to the shared state, it unconditionally deregisters all processes that have registered on it with an old sequence number. This is the same processing that would have otherwise been done had all the signals not been lost.

At least one "lost state" signal is late.

If a processor regains access to the shared state before one of the broadcast "lost state" signals is processed, processes that reregister on that processor will not be cleaned up when the signal is eventually processed. This is a direct consequence of the sequence numbering scheme outlined above.

"Lost state" signals are processed out of sequence.

In the unlikely event that a processor should lose, regain, and lose access again, it is possible that the "lost state" signals will be processed out of order. This case is covered by the sequence numbering scheme. The second signal, being processed first, results in all processes from both access instances being deregistered. This is because deregistration is done for all processes with sequence numbers less than or equal to the most recent access sequence number.

Access to state is lost before assigned processes are restarted.

Processes that have been assigned to another processor as a result of a processor failure but have not been restarted are deregistered if the assigned processor loses access to the shared state. This is accomplished by tagging processes with the assigned processor's current access sequence number as they are assigned. "Lost state" processing as described above will deregister these processes.

The processor that owns failure analysis loses access to state.

A signal is broadcast to relinquish ownership of failure analysis. This logically requeues the failure analysis event. Processes registered on this processor will be deregistered by "lost state" processing as described above.

Multiple processors lose access to the state.

Several processors losing access to the shared state "simultaneously" is handled by the general "lost state" method as described above. If all processors lose access, all processes in the entire system are considered logically deregistered. They will be removed from the shared state as each processor reaccesses it.

4 Cluster Issues

The discussion in the previous sections has been on the implementation of restart services in a tightly coupled environment. Similar services could be implemented in a loosely coupled environment (e.g., a distributed cluster of workstations).

Some of the problems mentioned in Section 2 take on a different flavor in a cluster environment. Failure detection and diagnosis generally become more complicated, since there are usually more components to deal with and it may be more difficult to trace the cause of the failure. For example, loss of communication between two nodes in a cluster may have resulted from failure of nodes, adapters, the network link, or even some bridge or gateway that needs to be crossed.

Distributed systems can involve many more processors than tightly coupled systems, since the latter are typically constrained by hardware. In a system with many processors, scalability becomes an issue, especially since communication and synchronization are computationally more expensive than in a symmetric multiprocessor. The scalability issue may render some approaches infeasible. For example, it may not be possible to have a single processor compute the configuration for the entire complex or the actions that need to be taken by everybody. Distributed algorithms are usually more difficult to write and debug.

The critical observation about the ability of processors to synchronize despite a failure becomes a bigger challenge in a loosely coupled system. Synchronization implies communication, and communication is usually the part of the system that exhibits most failures. The shared hardened hardware approach is not viable in a cluster. To compensate, it is necessary to build enough redundancy in the communication paths and enough logic to ensure that communication will be possible despite any (expected) failures. A similar argument holds about the availability of the policy data to make reconfiguration decisions in case of failure.

5 Conclusion

The paper has described a methodology for incorporating restart services into basic membership services. It is argued that *restart* is a fundamental building block for highly available systems. We have presented algorithms for failure detection and restart methods that have been implemented in a commercial system [7, 9, 8].

References

[1] Azagury, A., Dolev, D., Goft, G., Marberg, J., and Satran, J., "Highly available cluster: A case study," in *24th International Symposium on Fault-Tolerant Computing*, (Austin), pp. 404–413, June 1994.

[2] Bowen, N. S. and Pradhan, D. K., "Processor- and memory-based checkpoint and rollback recovery," *IEEE Computer*, vol. 26, pp. 22–31, February 1993.

[3] Cristian, F., "Agreeing on who is present and who is absent in a synchronous distributed system," in *18th International Symposium on Fault-Tolerant Computing*, (Tokyo), pp. 206–211, June 1988.

[4] Garcia-Molina, H., "Election in a distributed computing system," *IEEE Transactions on Computers*, vol. 31, pp. 48–59, January 1982.

[5] Gray, J. N., "Notes on database operating systems," in *Operating Systems: An Advanced Course* (Bayer, R. *et al.*, eds.), pp. 393–481, Berlin: Springer Verlag, 1979.

[6] IBM, *AIX High Availability Cluster Multi-Processing/6000 Version 2.1 Administration Guide*, December 1993. Document Number SC23-2599.

[7] IBM, *MVS/ESA Sysplex Overview*, December 1994. Document Number GC28-1208.

[8] IBM, *MVS/ESA Sysplex Services Guide*, December 1994. Document Number GC28-1495.

[9] IBM, *MVS/ESA Sysplex Services Reference*, December 1994. Document Number GC28-1496.

[10] Jahanian, F. and Moran, Jr., W. L., "Strong, weak and hybrid group membership," in *2nd Workshop on the Management of Replicated Data*, Novemeber 1992.

[11] Nelson, V. P., "Fault-tolerant computing: Fundamental concepts," *IEEE Computer*, vol. 23, pp. 19–25, July 1990.

[12] Peercy, M. and Banerjee, P., "Design and analysis of software reconfiguration strategies for hypercube multicomputers under multiple faults," in *24th International Symposium on Fault-Tolerant Computing*, (Boston), pp. 448–455, July 1992.

[13] Siewiorek, D. P., "Fault tolerance in commercial computers," *IEEE Computer*, vol. 23, pp. 26–37, July 1990.