# Availability in parallel systems: Automatic process restart

by N. S. Bowen
J. Antognini
R. D. Regan
N. C. Matsakis

***Parallel and clustered architectures are increasingly being used as a foundation for high-capacity servers. At the same time, the availability expectations are also rising rapidly, since the effects of down time become more apparent and have higher economic consequences for larger systems. The use of parallel structures generally implies more hardware and software components. The presence of more and larger components increases the chances that an individual component will fail, and that failure has the potential to hurt the overall availability of the system. This paper discusses the use of "restart techniques" as an important strategy in providing increased availability in a parallel structure. The paper covers a set of functions that have been developed for the S/390® Parallel Sysplex™.***

Nearly all users of computer systems are making availability a *de facto* requirement. Also, a strong demand for higher performance is increasingly being met with clustered architectures.[1] The redundancy inherent in clustered systems offers the opportunity to provide increased levels of availability. However, the presence of more and larger components increases the chances that something will go wrong and tends to decrease availability. The ability to rapidly detect and recover from component failures is a critical function for providing high availability in clustered systems.

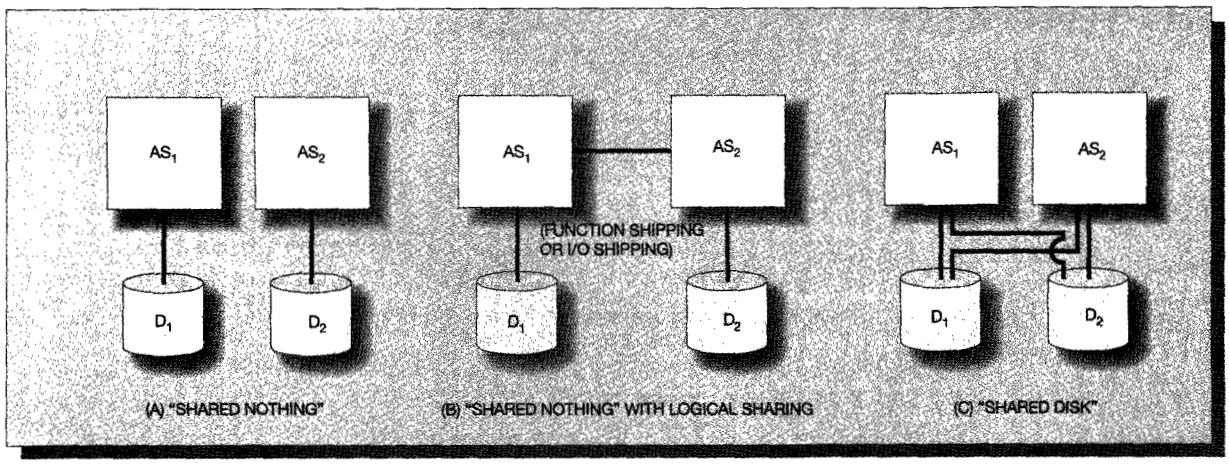
An *application server* is defined as a collection of hardware and software that can run a specific application (e.g., a transaction, database query, an editor). Furthermore, *application availability* is defined as the probability that the application server is available at a given point in time. Clearly, clustered systems have an advantage in providing increased availability because of the inherent redundancy in the system. However, the design of the application server with respect to the various failure modes of the system dictate the overall availability. For example, if the application server can continue to provide services even when an individual system fails, it can have greater availability than a corresponding server on a single system.

Pfister describes many clustered architectures that have the basic objective of using the underlying architecture to provide increased availability.[1] Although many of these systems have implemented heartbeat mechanisms for detecting system failures, he observes that quite a few systems use "fail over" techniques that depend on application or system restart techniques because the application server itself must be restarted to provide continued service. Such use is due to the underlying "shared-nothing" architecture of these systems where data are often partitioned among the various processing nodes. That architecture can be contrasted with the System/390* (S/390*) Parallel Sysplex* that is based on a "shared-disk" architecture where the application servers have access to all data and are capable of running any application. Here the requirements




0018-8670/97/$5.00 © 1997 IBM

**Figure 1 Application server architectures**




differ; restart is necessary for restoring the initial configuration and performing recovery actions (e.g., database log recovery). We claim that a set of basic operating system restart services are required for both of these scenarios. From this claim we assert that three principal steps are required in obtaining high availability in a clustered system; namely, fault detection and containment, the ability to operate in *degraded* mode, and finally, the ability to restore the original configuration.

This paper examines a set of services intended to facilitate this task. First, we define the system model and present the requirements for high availability, then we describe the restart services and some of the robust features of these services. A preliminary version of this paper can be found in Reference 2.

## Availability strategy and system model

This section provides a high-level overview of the S/390 Parallel Sysplex and outlines the overall strategy for providing highly available application servers. First, the impact of the general architecture for parallel systems is discussed.

The architecture of the *application server* is critical to understanding the requirement for restart processing. Figure 1 shows three classic approaches used in clustered systems. A shared-nothing architecture is the basis for cases A and B in which the disks are attached to a single system. Case A shows a fully partitioned approach in which particular application servers can only access a subset of the data and, thus, can only run a subset of the applications. Case B shows a basic enhancement to the shared-nothing approach that allows an application server to access remote nodes using techniques such as "function shipping" or "I/O shipping."[1,3,4] In case C, all systems have access to all data. One could also imagine additional variations (e.g., an application server with the properties of A running on a parallel architecture as defined in C). From these various scenarios one can discern various reasons for using restart services. These include:

1. Restart of the application server
2. Restart to recover resources (variation of peer recovery)
3. Restart to restore configuration

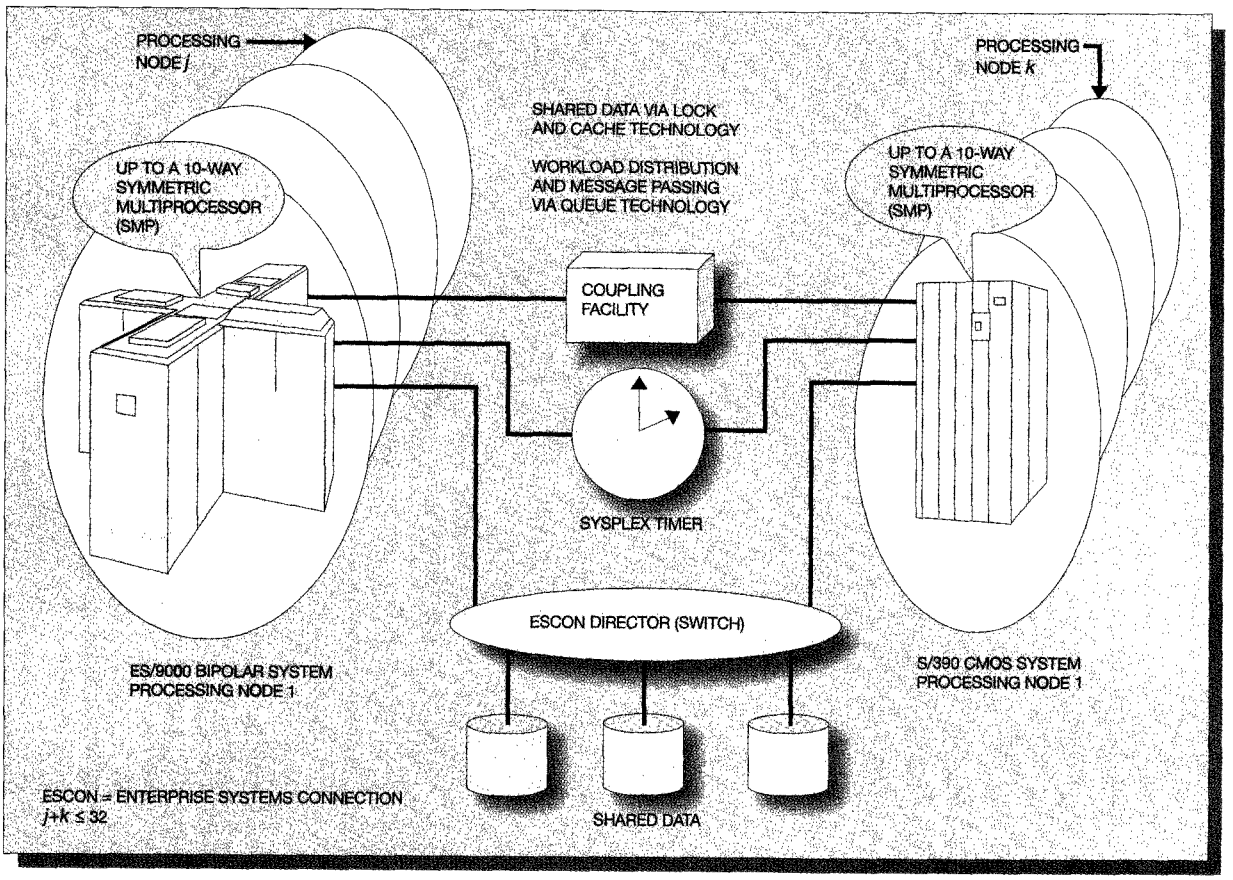
Before discussing the specific restart scenarios, basic transitions that the system must go through are discussed. Furthermore, it is argued that the following steps must occur for all scenarios:

1. Failure detection and isolation
2. Continuous operation in "degraded" mode
3. Restoration of the initial configuration

The duration of each step and the definition of *degraded* differs for the various scenarios. These steps are now described in more detail.

There are several components of failure detection and isolation: first, the use of heartbeat mechanisms

**Figure 2 Parallel Sysplex system model**

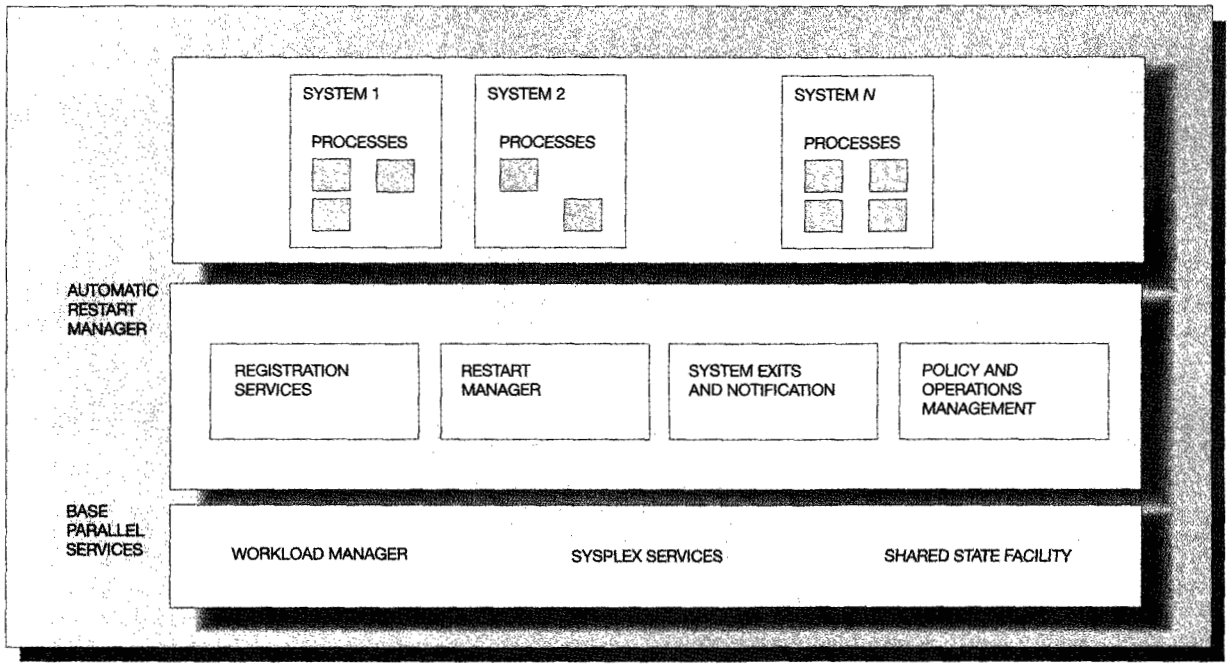



to detect an unhealthy system and, second, the ability to physically *partition* the failed system out of the cluster. That is, the system itself is disconnected from its own I/O processor, and all required components are notified of the failure of the system and all processes on that system. Finally, the workload scheduler components are made aware of the failure and continue to route work into the cluster while avoiding the failed region.

For these first two cases (from Figure 1), the key objective is restart of the application server, which may also require a restart of the system. The application server is unavailable for a subset of the applications (thus, one definition of "degraded") during the restart process. In case C, the new requests can be routed to other application servers immediately, thus maintaining availability. The key restart objective is the initiation of the application server recovery logic (e.g., database log recovery). Here the term "degraded" means lower capacity, but the applications can continue to run.

Figure 2 shows the overall structure of the S/390 Parallel Sysplex.[3,4] It consists of up to 32 processing nodes (each node can be a symmetric multiprocessor with 1 to 10 processors), each running the OS/390* operating system and connected to a collection of shared disks. The basic system design has a long history of fault-tolerant features.[5] The I/O architecture has many advanced availability and performance features (e.g., multiple paths with automatic reconfiguration for availability). The basic I/O architecture is described in Reference 6, and one aspect of the dynamic I/O configuration is described in Reference 7. The Sysplex Timer* (ETR) serves as a synchronizing time source for systems in the sysplex, so that local system time stamps can be relied upon for consistency with respect to time stamps obtained on other systems. There is also a facility called XRF (extend-

**Figure 3 Overall system structure**




ed recovery facility) that allows for "hot standbys." The coupling facility (CF) provides multisystem data-sharing functions and is described in References 3 and 4.

## Automatic restart manager

Since all systems in the Parallel Sysplex can have concurrent access to all critical applications and data, the loss of a system because of either hardware or software failure does not necessitate loss of application availability. Failing applications, caused by system or process failures, can be automatically restarted on still-healthy systems by the OS/390 automatic restart manager (ARM) component to perform recovery for work in progress at the time of the failure. While the failing application server instance is unavailable, new work requests can be redirected to other data-sharing instances of the server to provide continuous application availability across the failure and subsequent recovery. ARM is fully integrated with the existing parallel structure and provides significantly more functions than does a traditional "restart" service. First, utilizing a shared state support facility[3,4] at any given point in time, ARM is aware of the state of all processes on all systems (including processes on any failed systems). Second, ARM is tied into the system heartbeat functions so that it is immediately aware of system failures. (In a subsequent section, this notion of "immediate" is clearly defined by carefully describing the actual time sequencing of restarts.) Third, ARM uses the workload manager to determine a target restart system based on the current resource utilization across the available systems. Finally, ARM has many features to provide improved restart such as affinity of related processes, restart sequencing, and recovery when subsequent failures occur. Figure 3 shows the overall software structure.

A set of base parallel services provides many of the clustered services (e.g., membership services) that are not described here, but a summary can be found in References 3, 4, and 8. The functions for ARM are described in several sections. First, the operating system services and the general usage of ARM are described. Second, *restart manager* functions and algorithms are described, including a discussion of the fault-tolerant features of the algorithms. Next, the operational aspects of ARM are covered. Finally, the usage of *system exits* and the *event notification facility* to communicate special asynchronous events are

**Table 1 Input parameters for register service**

| Parameter | Used to Identify |
|---|---|
| Name | The specific process |
| Type | A generic class of processes |
| Event exit | A program to run prior to restart |
| Event exit data | Parameters for the event exit |
| Restart command | An alternate restart method |
| Failure type | Actions for various failure types |

**Table 2 Output parameters for register service**

| Output | Used to Indicate |
|---|---|
| Restart type | If process was restarted |
| Restarts enabled | If restarts are currently enabled |
| Prior system ID | Previous system the process was on |

**Table 3 Summary of commands**

| Service | Used to Indicate |
|---|---|
| Register | Restart services are requested |
| Deregister | Restart services are not required |
| Ready | Restarted process has completed restart logic |
| WaitPred | Process will wait for dependent processes |
| Associate | A hot-standby alternate is active |

described. The formal programming interfaces are defined in Reference 9, and a description of the usage can be found in Reference 10. Although the focus of this paper is on restarts in parallel systems, it should be noted that the ARM services also apply to the situation where a process fails and is restarted on the same system.

**Basic restart services and terminology.** A *Register* service is provided to allow a process to indicate when restart services are required and a *Deregister* service to indicate when they are no longer required. The parameters used on the register service are shown in Table 1.

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* indicates whether the operating system is currently enabled and capable of performing restarts. *Prior system ID* provides an identification of the prior system on which the process was executing.

The full set of services are summarized in Table 3. The *Ready* and *WaitPred* services are used to synchronize processes that have sequencing dependencies.

A process that is using ARM services is referred to as an "element." In addition, we define a *restart group* as a set of elements that have affinities and must remain together in the event of a system failure and resulting restart. Furthermore, the restart group has an effect on the sequencing of elements (described shortly). Elements are not aware of being in a restart group; the elements are placed in a group based on the element name and a system administrator's policy.

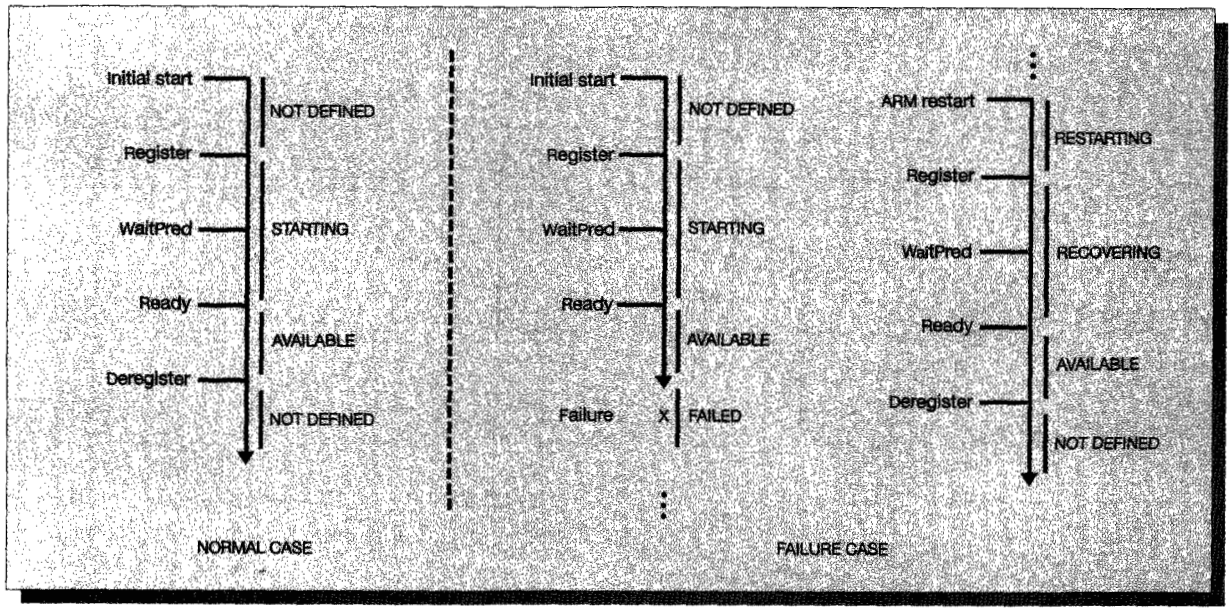
ARM also provides commands to allow sequencing of restart activities. These activities include the services WaitPred (Wait for Predecessors) and Ready. These services are dependent on a concept called "restart levels" wherein elements can be arbitrarily placed in numbered levels. The requestor of the ARM service is unaware of its level; it is set by the installation. The WaitPred function is provided because elements may depend on other elements being initialized and available.

The Wait for Predecessors service does not cause a wait if this element is not part of a cross-system restart. So, for example, nothing really happens for an element starting initially (without ARM intervention, that is) or being restarted within the same system by ARM.

In cases where an element is to be a predecessor for other elements in a cross-system restart, that element has an obligation to make known its readiness for work. This is done by the Ready service. As with the Wait for Predecessors service, there are no parameters. It is up to ARM to keep track of what elements may be waiting for this element to say ready. In cross-system restart, a Ready request will not complete until all elements with lower levels in the containing group indicate that they are ready. In other words, Ready behaves in such a restart as though it had been preceded by a Wait for Predecessors request.

ARM provides special restart support for hot-standby environments. In a hot-standby environment, you have a primary server ($P_p$) and a backup server ($P_a$). The recovery concept in this environment is that in the event of a failure of $P_p$, then $P_a$ immediately takes over the processing. These two systems are very

**Figure 4 ARM activity and states (normal case to left, failure case to right)**



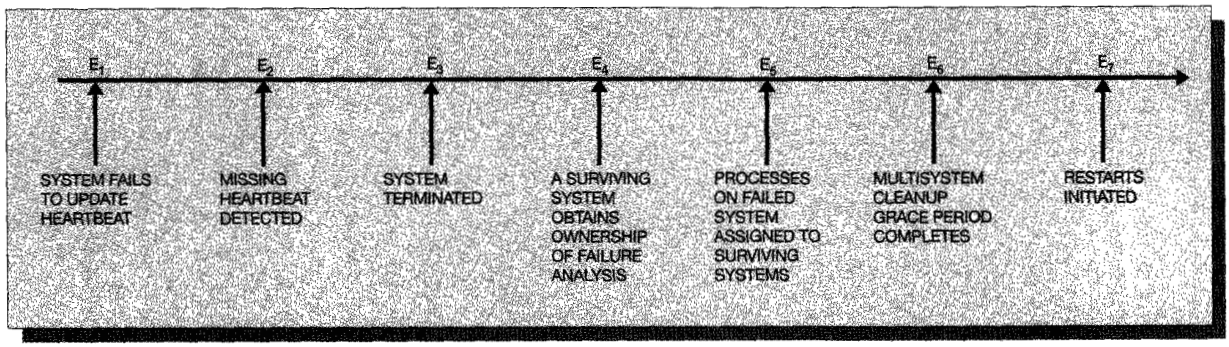

tightly integrated so that they typically do not have to do recovery functions such as log recovery (because the backup will have been monitoring the logs). With regard to restart, process $P_p$ does not need to be restarted to provide availability; in fact, it was deemed that a restart of $P_p$ would be undesirable because it would unnecessarily consume resources. Thus, ARM provides an *Associate* service that allows an element to indicate that a certain other element is not to be restarted by ARM. ARM does not, however, inform this element when the other element terminates. The responsibility for knowing that lies with the element imposing association. In contrast, if the associating element itself terminates, ARM does undertake to reinstate the other element's eligibility for restart (i.e., the Associate is removed).

A set of services on OS/390 allows for traditional multisystem membership services (e.g., join a group, signal members of the group).[8] ARM has provided the ability for members of multisystem groups to be informed of system failures and for them to effect a delay of the restart processing. This function is provided to allow these multisystem groups to perform special processing (e.g., resource cleanup) before the restarts occur. There is an optional parameter on the group Join service to request this function. Once notified of this situation, the member is expected to apprise ARM that it has finished cleanup and preparatory actions relating to the failure of the system. ARM will not proceed with restarts until that member and all other group members that have requested cleanup responsibility have declared that they are finished with cleanup (or until a certain amount of time has passed after they were notified of the failure). Failure of a group member is taken to mean that the member has discharged its cleanup responsibility.

At this point we describe the various states that an element can be in with respect to ARM. The set of ARM states is shown graphically in Figure 4. Depending on what ARM services an element has used and on the execution history of the element, an element can have one of several states:

- *Starting*: The element is executing and has registered.
- *Available*: The element is executing, has registered, and has indicated that it is ready for work.
- *Available-to*: The element is executing after being restarted, has registered, and is considered ready (available due to time-out) since it did not signal readiness in a certain span of time. This state is equivalent to *Available* (and thus not shown in Fig-

**Figure 5 Major epochs in restart processing**




ure 4) with respect to the restart sequencing except that the Ready call was not made.

- *Failed*: The element is registered, terminated (abnormally or normally) without first deregistering, and is waiting for the ARM restart process to commence.
- *Restarting*: The element failed, and the ARM restart process began and is still underway or is complete. If the restart process is complete, job scheduling factors could delay or even preclude execution. Otherwise, the element is executing but is yet to register again with ARM.
- *Recovering*: The element is executing because of a restart, has registered, but is yet to indicate that it is ready for work.

**Restart manager.** This subsection describes the specific algorithms used to perform the restart processing. The major epochs in the restart processing are described, using Figure 5 as a reference.

At epoch $E_1$, a system-level heartbeat mechanism is used. This first event indicates the first instance in which a system fails to update its heartbeat.

At $E_2$ the missing heartbeat is now detected. The delay from $E_1$ to $E_2$ 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 system to miss a heartbeat. The term "predatory takeovers" is used to describe the situation where the heartbeat mechanisms indicate a failure has occurred when, in fact, no failure actually occurred. Therefore, it is important to understand the sensitivity of the heartbeat mechanism to transient conditions and adjust the "missing heartbeat" parameters accordingly.

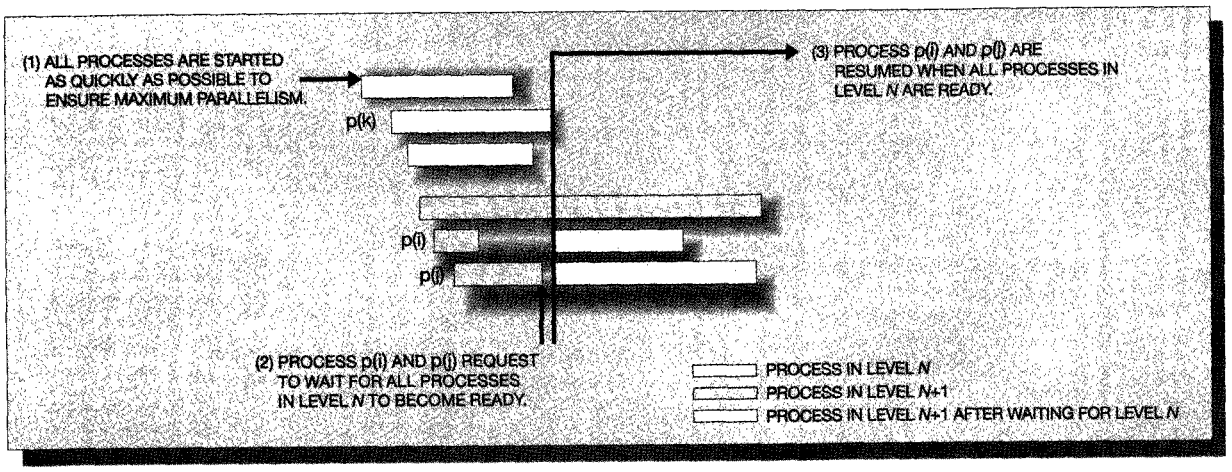
Starting at epoch $E_3$ actions must take place to ensure that the system is in fact terminated. This assurance is required because the imperfect nature of some heartbeat mechanisms can present the appearance of a failure. For example, a system could stop updating its heartbeat because of a loop in the operating system. The missing heartbeat might then be externally recognized, and restart processing would occur; then an event occurs that terminates the loop (e.g., an I/O error condition cleared, time-out logic in the operating system). The system then comes back to life, and there could be two copies of a process, one executing on the apparently failed system and one elsewhere. Two copies could cause database problems in a shared-disk environment or network problems in a distributed system.

ARM relies on the use of a technique that we refer to as "I/O fencing" in which there is a special communications link to the I/O mechanism of the failed system that allows I/O activity from the failed system to be disabled.

At $E_4$ a single system must become the owner of the restart process for the failed system. This determination could use any algorithm[11] for either tightly coupled or clustered systems. Our approach is to use a shared disk that uses a disk-based locking scheme. The system that becomes the logical owner records its system identifier on the shared disk.

At $E_5$, once a system becomes responsible for the restart process, it begins a set of steps that lead to assigning the processes to execute on other systems and initiating their restarts:

**Figure 6 Parallelism in restart processing**




1. Assign each process to a restart group of processes that must always remain together.
2. Assign each restart group to an available system:
   - Honor any static affinities.
   - Assign the restart group to a system based on recent historical resource consumption (e.g., CPU, memory) on the active system.

At $E_6$ the system provides a period of time for all members of multisystem groups to perform multisystem cleanup before the restarts are initiated.

At $E_7$ the system controlling the restart process (event $E_5$) signals all other systems to initiate the restarts. 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 system determines which processes must be restarted, the following steps occur:

1. If the process specified an event-exit, execute the program.
2. For all processes in a given restart group, create a list specifying the order in which processes must be restarted. If multiple processes have the same level, they are logically started simultaneously.
3. Initiate the restart of all processes in the restart group using a simple pacing algorithm that specifies delays between process restarts.
4. Enable sequencing of processes within a restart group. The processes have Ready and WaitPred calls that allow them to wait for processes that were started earlier (i.e., that have a lower level).

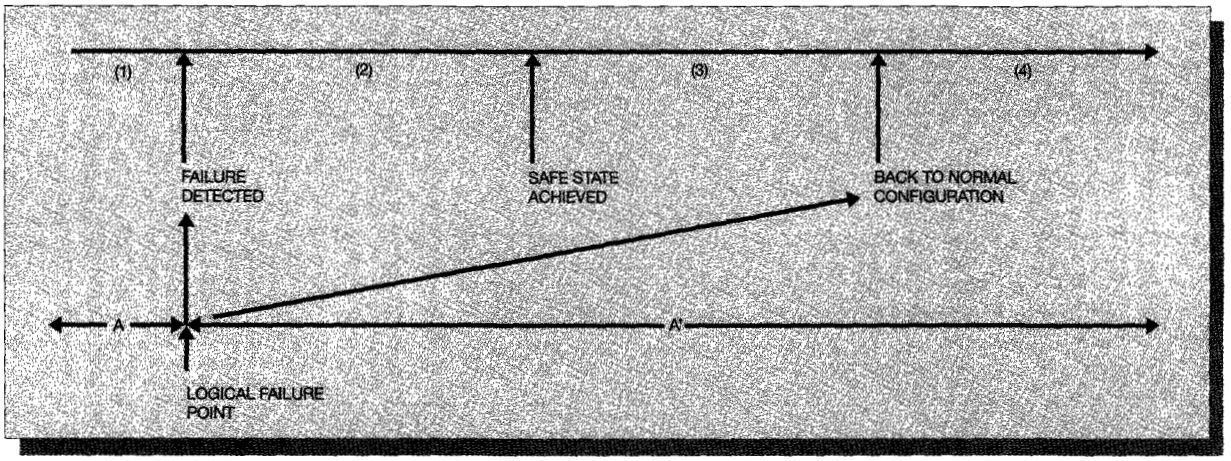
The key aspects are illustrated in Figure 6. In addition, all processes within a restart group can be restarted with a pacing value that staggers the restart processing to avoid a system overload. Furthermore, the synchronization is only done for process threads that issue the ARM services. That is, other threads under the restarted process are not affected.

Whether and how a registered element that terminates is to be restarted depends on a number of things:

- If an operator cancelled the element (using a CANCEL or FORCE command), the element will not be restarted. However, the use of the optional parameter ARMRESTART will cause the normal termination followed by a restart.
- The element has not been restarted more than a fixed number of times within a fixed time window.
- No other element has made an Associate request against this element.
- An element-restart exit (defined later) allows restart. The exit may also change the manner of restart.
- Any event exit named in the registration of the application allows restart. This exit cannot, however, change the manner of restart.

**Fault tolerance in algorithms.** One of the key values of ARM is the notion of logical atomic failure

**Figure 7 View of failures**




points created during the complicated underlying physical state transitions that actually occur during system failures and subsequent restart processing. Whereas the previous subsection described the basic processing that the restart manager applies to system failures and restarts, this subsection focuses on the underlying design principles and uses a few examples of very complicated failure scenarios to illustrate the points.

Since we wish to provide operating system services for building highly available systems, we must ensure that the services themselves have additional fault-tolerant features. A methodology for tolerating faults by precisely defining the way in which failures are exposed to users of the services is now formulated. Our design objective is a system that can recognize failures, remain operational while failures persist, and restore itself to normal operation when failed components are repaired. A set of physical states and logical states that occur because of a failure are described. The physical states are:

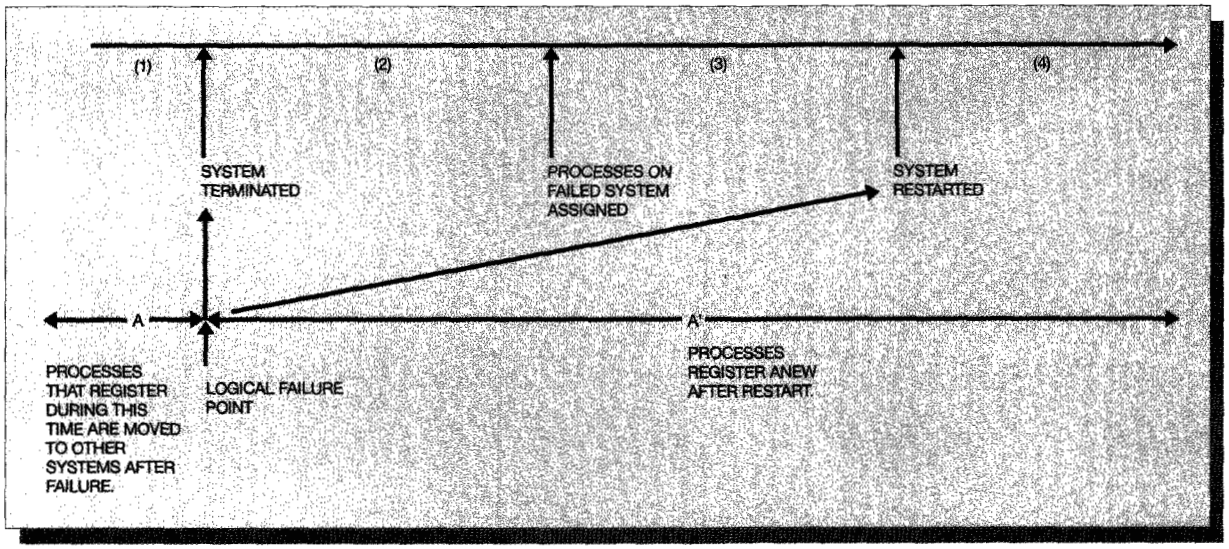
1. Normal operation
2. A period of time from when the system has recognized the failure of one of its components to when it restores itself to a limited, yet safe, configuration
3. A period of time from when the system establishes a safe configuration to when it reestablishes its original configuration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

Failure masking is a key objective in this design. Therefore, we mask the physical state transitions just described and present a logical view to the user of the restart services. Masking is illustrated by the lower time line in Figure 7. 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 register 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 in different places at different times.

Achieving this logical view mandates additional design to handle events that occur when the underlying system is repairing a physical failure. Solutions include techniques such as queuing the ARM requests (or rejecting them) and notifying users upon repair (essentially queuing in the operating system or queuing in the user). Following are two specific examples of the basic fault tolerance model.

*System termination.* When a system fails, two fundamental actions must occur. First, the processes from the failed system must be restarted on active systems. Second, the system itself must be restarted. It is desirable to perform these two activities in parallel. However, this desire creates the possibility that the

**Figure 8 Realization of system failure**




system can be restarted before the elements have been restarted. It means that new elements could register on the newly restarted system and then appear as if they were executing on the failed system. The notion of logical failure points is used to make this scenario impossible. This notion not only makes the implementation easier and more efficient, but also less complex so that it becomes more reliable. If these two activities can be performed in isolation from each other, the implementation is greatly simplified. However, if these activities are independent, one must be careful regarding processes that register after the system is restarted (i.e., they should not appear as candidates for process restart).

Figure 8 is used to illustrate the application of this methodology to a system failure scenario. As described previously, a system failure results in the processes being assigned to another system. With our design methodology, the overall design is simplified 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 system restarted and thus should not be considered failed). It is important to note that this discussion applies to the internal design of the restart manager and does not have implications with respect to multiple "instances" of processes across these logic points (i.e., if a process is erroneously restarted while a prior incarnation is awaiting to be restarted, there will be an unwanted duplicate).

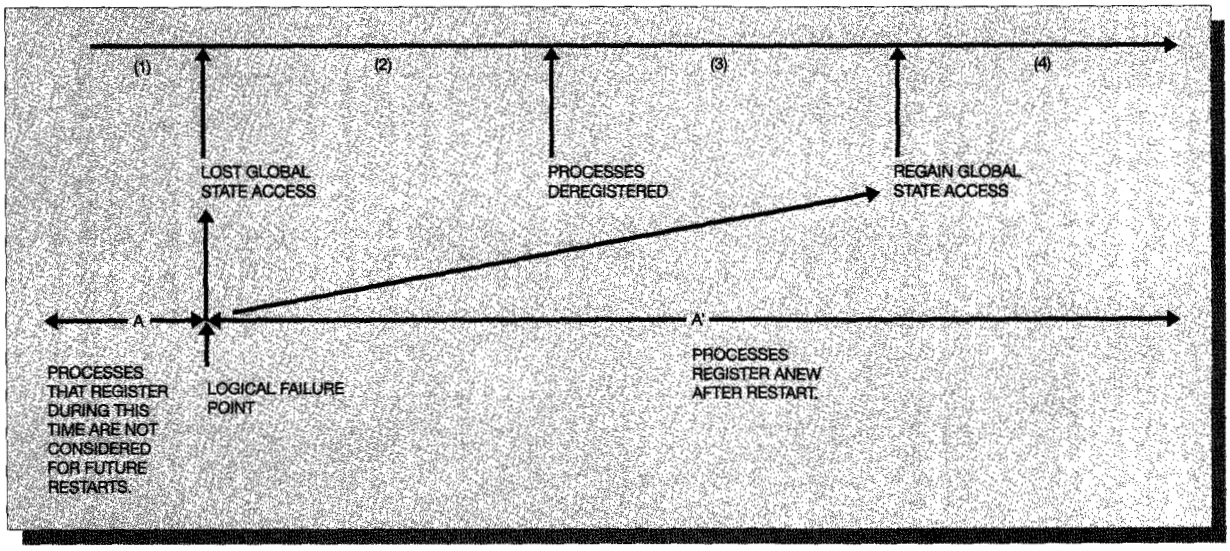
*Loss of access to the shared-disk global state.* As we stated earlier, the information required to determine the necessary reconfiguration must be accessible in case of failure. This information is kept on a shared disk, with independent access from every system to the disk. To guard against failures, 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 system 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:

1. A process deregisters itself, but the system cannot access the shared disk to record the deregistration.
2. The system on which the process is running terminates, and the process is (incorrectly) restarted on another system.

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

**Figure 9 Realization of global state failure**




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 and deregistration actions could be sent to other systems and recorded on disk by them. For resiliency, the actions would have to be sent to many (all) other systems, and it would be necessary to arbitrate among those systems 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 system of origin (the one that lost access to the disk) is preserved. For example, the relative order of two registration or deregistration events should not be reversed by messages received out of order by other systems.

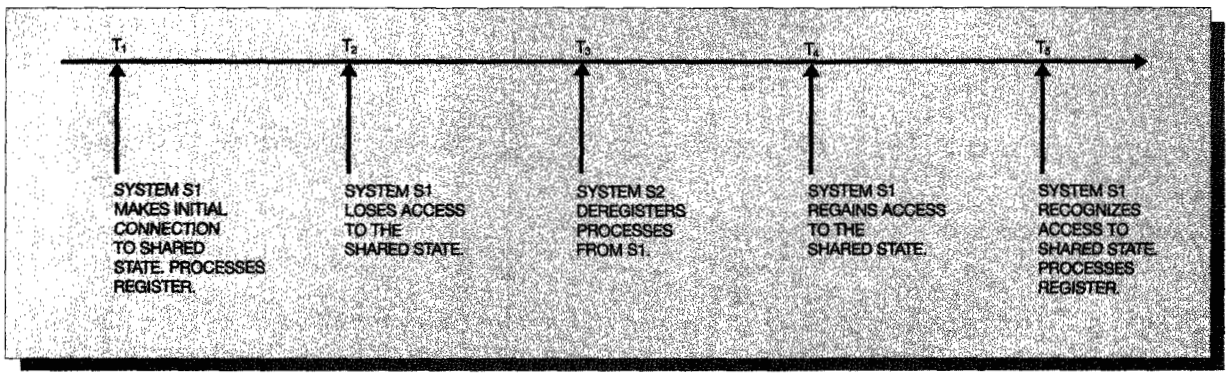
Implementing a message-based protocol to tolerate the loss of access to the shared disk by one system would be a difficult effort. The gain in service availability would be minimal, because the loss of access to the shared disk is a very rare event, given that it requires at least a double failure (single failures are masked by the redundant hardware used).

We have opted for a solution that brings the system to a safe state when a system 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 system (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 system; that is, processes are logically deregistered, which prevents undesirable restart scenarios like the one outlined above.

Figure 9 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 systems that perform the deregistration remove only those processes in set $A$ and do not touch processes in set $A'$. Below are described 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. Two sets of sequence numbers are defined:

1. An array of sequence numbers SysSeqNum(), which contains one sequence number per system. Each sequence number SysSeqNum(S) repre-

**Figure 10 Major epochs in recovery of global state**




sents the $n$th connection to the shared state by system S.

2. An array of sequence numbers ProcSeqNum(), which contains one sequence number per registered process. Each sequence number ProcSeqNum(P) is a copy of SysSeqNum(S) as it existed at the time process P registered on system S. This represents the bind between a process and the particular configuration under which it registered.

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

An example, depicted in Figure 10, showing the steps involved in recovering from a loss of access to the global shared state is now described.

At $T_1$, system S1 makes its initial connection to the shared disk. SysSeqNum(S1) is initialized to one (i.e., initial access). When process P1 registers on system S1, its state data representation is tagged with the access sequence number of S1. That is, ProcSeqNum(P1) is set to SysSeqNum(S1).

At $T_2$, an attempt to access the shared state fails. A "lost state" signal containing the logical system identifier and SysSeqNum(S1) of S1 is broadcast to all other systems to request that processes registered on system S1 be deregistered. Any system with access to the shared state is eligible to perform the deregistration process on behalf of system S1.

At $T_3$, system S2, which still has access to the shared state, receives the signal from system S1 and logically deregisters all S1 processes that are tagged with a sequence number that is less than or equal to the sequence number in the signal. Included in the deregistered processes are processes that originally registered on S1 and processes that have been assigned to S1 but have not yet been restarted. (When processes are assigned to a target system, they are tagged with the current access sequence number of that system.)

System S2 also updates the state information to indicate that system S1 no longer has access to the shared state. In addition, each affected process on system S1 is notified that it has been logically deregistered (assuming that it is "listening" for this event through the Event Notification Facility).

At the $T_4$ epoch, at some time in the future, the path to the shared state from system S1 is repaired, and system S1 regains access. Before reaccess is officially recognized, system S1 attempts to deregister all processes that may still be registered with a previous SysSeqNum(S1) (the same processing that is performed in step $T_3$). This step is taken as a precautionary measure in case the "lost state" signals are lost or delayed.

At $T_5$, system S1 increments SysSeqNum(S1) (i.e., its value is now two) and updates the state data to indicate that it now has access to the shared state. Notification is broadcast (through the Event Notification Facility) to all "listening" processes on S1 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 system fails). These algorithms tolerate the following timing anomalies:

- All "lost state" signals are lost. When a system regains access to the shared state, it unconditionally deregisters all processes that have registered on it with an old sequence number. This processing is the same as would have otherwise been done had all the signals not been lost.
- At least one lost state signal is late. If a system regains access to the shared state before one of the broadcast lost state signals is processed, processes that reregister on that system will not erroneously be cleaned up when the signal is eventually processed. It 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 system 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 the deregistration of all processes from both access instances 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 system as a result of a system failure but have not been restarted are deregistered if the assigned system loses access to the shared state. Since processes are tagged with the current access sequence number of the assigned system, lost state processing as described above will deregister these processes.
- The system that owns failure analysis loses access to state. A signal is broadcast to relinquish ownership of failure analysis, which logically requeues the failure analysis event. Processes registered on this system will be deregistered by lost state processing as described above.
- Multiple systems lose access to the state. The case of several systems losing access to the shared state "simultaneously" is handled by the general lost state method as described above. If all systems lose access, all processes in the entire system are considered logically deregistered. They will be removed from the shared state as each system reaccesses it.

**Policies and operational management.** Policies are one way for an installation to exercise control over how and even whether an element is restarted. For in-place restarts of an element, control takes the form of setting a limit (possibly zero) to the number of restarts that will be allowed or of specifying an overriding method of restart.

For cross-system restarts, a variety of additional things can be done. The installation must not only control restarts and thereby the availability of service, but also must aim for adequate load balancing. A surviving system must not be so overburdened with restarted elements that inadequacy of service results.

A second thing to consider in cross-system restarts is a suitable sequence and pacing of restarts, so that a restarted element will find those services it needs available. In the normal case the availability of services is ensured by the operational procedures of the installation, but in cross-system restart ARM must substitute for those procedures.

The *ARM couple data set* is the repository of the ARM policies of an installation and also of the specifics of elements with ARM status. This data set is separate from other couple data sets (e.g., those of workload manager), may have an alternate data set, and must be connected to all systems where registration and restart might occur. The *ARM policy* is a set of instructions from an installation about how and where (and whether) restarts are to be done. The main purpose of a policy is to define the elements comprised by a group, with particulars about dependencies in the group, overriding sources of restart techniques and parameters, selection criteria in cross-system restarts, and pacing of restarts.

Following are some of the parameters that a policy can comprise:

- RESTART_GROUP (*name*)—Specifies the elements that are to be restarted together in cases of failure of the system on which they are running. Subparameters are:
  - ELEMENT (*list of elements*)—A list of one or more names of elements constituting the group.
  - TARGET_SYSTEM (*list of systems*)—A list of one or more systems on which a group will be restarted. If this is not specified, all systems will be eligible. ARM will employ workload manager to determine the most suitable system in the target set.
  - FREE_CSA (*size of memory available*)—This subparameter allows one to mandate that for a sys-

tem to be considered a target of the restart processing it must have a minimum amount of *common service area* memory available.
- RESTART_PACING (*number of seconds*)—The number of seconds for ARM to wait between restarting one element in the group and restarting the next. The default is zero, which causes all the elements in a group to be started in parallel. Thus, the primary effect of this parameter is to specify how many elements in a group are restarted in parallel. But whether the elements are restarted in parallel or serially, the order of restart will be unpredictable.

The next two subparameters apply to in-place as well as cross-system restarts.

- RESTART_ATTEMPTS (*maximum restarts, number of seconds*)—The maximum number of restarts of an element to be done within a given time period. Specifying zero prevents any restarts. Once an element exceeds its limit, it is deregistered.
- RESTART_METHOD (*event type, overriding command for restart*)—Overriding technique for performing the restart.

- RESTART_ORDER—Controls the order in which elements in a given group are to be made to wait (when they request Wait for Predecessors) for other elements in the group to indicate that they are ready. An element has a level (if only by default), and elements at a given level will wait for elements at lower levels.
- LEVEL (*n*)—This is the level associated with one or more specified element names.
- ELEMENT_TYPE—This parameter allows generic mappings (e.g., assign a level) for a class of elements. It is based on the "Element_Type" parameter on the Register service.

ARM supports the notion of duplexed coupled data sets for improved availability. A set of operator commands allows dynamic reconfiguration of these data sets, including the ability to dynamically add or delete a backup data set, switch the primary and backup, as well as activate or terminate policies.

**Exits.** In OS/390 an "exit" is a programming method in which a component of the operating system can be tailored by code added by the customer. The reader is referred to Reference 12 for a description that is beyond the scope of this paper. For the purpose of this paper, exits are the place where an installation or an application can exercise programmatic control over the behavior of the automatic restart manager to effect a general-purpose restart structure. The relevant exits are:

1. Group exit—For each member of a multisystem group (i.e., using the Join service) there can be a group exit. A group exit gets control, among other conditions, when a system that is part of the sysplex fails. If the member requested this option, ARM will wait (for a reasonable amount of time) for the member to perform actions appropriate to it and to invoke the IXCSYSCL service to signal that it has completed those actions.

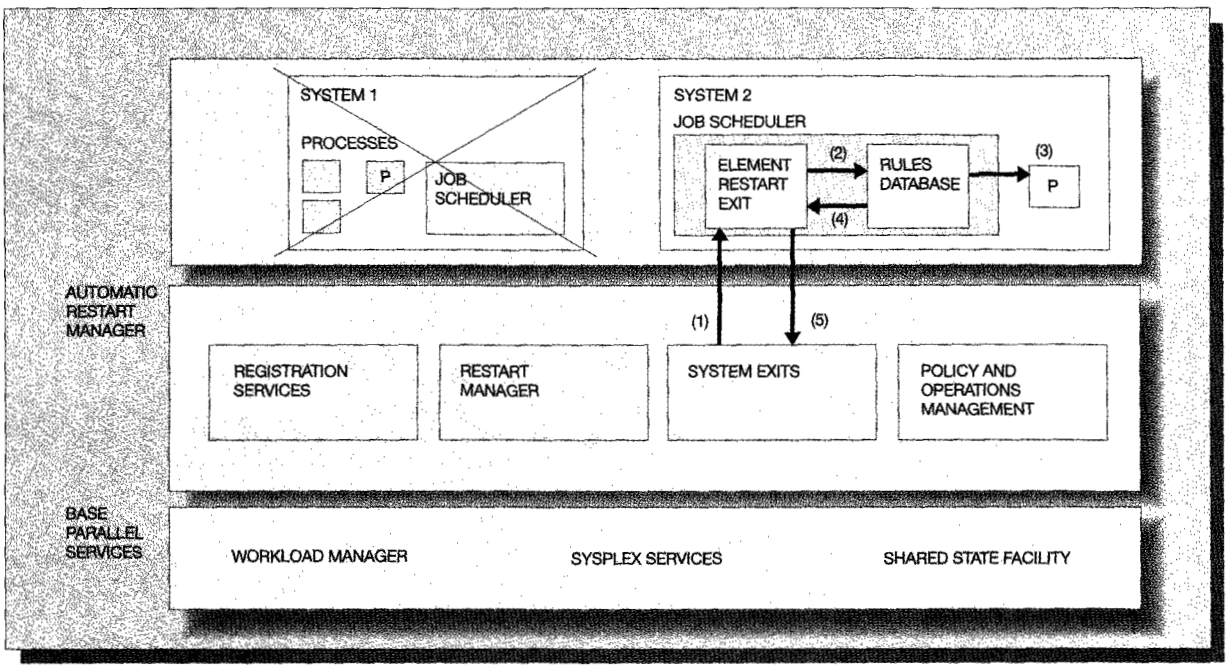
2. Workload-restart exit—In cases of a failed system with ARM elements, any workload-restart exit for a system where one or more groups will be restarted has an opportunity to perform cleanup and preparatory functions. This exit runs after all group member cleanup-complete requests have been issued. ARM will not restart elements on this system until the workload-restart exit has run (or until a reasonable amount of time has elapsed). The exit will have information about the name of the failed system and about the elements that will be restarted on this system. The member cannot, however, veto restart and cannot directly affect the manner of restart.

3. Element-restart exit—An element-restart exit is a generic exit that gets control whenever any element is to be restarted both on the same system and across systems. The exit has the options of vetoing the element's restart and of allowing the element's restart to proceed without change or with changes. If the manner of restart is to be changed, the exit can specify different parameters for restarting the element than were used for the prior start or specified at registration. It can also restart the element and then notify ARM of this action.

4. Event exit—An event exit is an element-specific exit that gets control when a particular element is to be restarted. This exit has the choice of allowing restart to occur or of vetoing restart (but the exit cannot change the manner of restart).

5. Event Notification Facility (ENF) exit—This facility, a part of OS/390, provides two complementary functions.[13] First, software components can broadcast the occurrence of "events." There is a prearranged scheme for identifying specific events. Second, software components can "listen"

**Figure 11 Extensibility of the restart manager**




for specific events, requesting that the operating system notify them when a given event occurs.

When certain events occur, ARM will cause an ENF signal to be generated.[14,15] The specified exit will get control in these cases:

- An element did one of the following: registered, reregistered, said it is ready, or deregistered.
- Deregistration occurred because of internal error.
- Restart of an element failed.
- Access to the ARM couple data set has been lost or regained.

With this information, an application will be able to know, for example, whether an element actually succeeded in being restarted. Or if access to the ARM couple data set is regained, the exit may inform an application that it can now register with ARM.

A significant amount of software has been written to control the scheduling of work for large systems. For simplicity, we use the term "job scheduler" as a generic term to describe this software. Two key goals were considered for this software:

1. Provide appropriate real-time signals, programmable interfaces, and operational controls so that a job scheduler could coexist with ARM.
2. Provide an extensible structure so that a job scheduler could use ARM to easily become a multisystem job scheduler.

There are a large number of extremely complex functions in ARM that would be very difficult to reproduce at a level above the operating system. These functions include system failure detection, target system selection and the integration with the workload manager, and many of the subtle timing and sequencing issues.

An example is now shown of how a job scheduler could leverage the ARM services to become a multisystem job scheduler without having to write a large amount of code. Figure 11 shows a scenario in which a job scheduler is running on multiple systems, and System 1 has just failed. ARM performs all the functions described thus far and selects System 2 as the target system for the restart. Since the job scheduler has defined an *element restart exit* (flow 1), ARM notifies the job scheduler that a restart is about to occur. In this example, the job scheduler looks up its local rules database and determines that this pro-

cess should be restarted. It then restarts the element (flow 3) and returns to ARM in flow 5. In the return parameter list to ARM, it indicates that the restart was done by the job scheduler, and ARM uses that information to update its state. It is important to note the exits are running on the selected target system.

Many other interactions are possible with the job scheduler, including variations on the above scenario or uses of the other exits that have been defined.

## Summary

This paper focused on high availability in parallel or clustered systems and on the need for restart services as a basic building block. Algorithms for failure detection and restart methods were described that have been implemented on OS/390.[16] One of the key principles is that the restart service presents the notion of logical atomic failure points in order to shield the system from the very complicated events that occur during a system failure and subsequent restart processing. In addition, the restart service provides flexible controls to support a general-purpose restart structure.


## Acknowledgment

We would like to thank Dave Petersen and Jim Daly for their contributions to the design of ARM.


*Trademark or registered trademark of International Business Machines Corporation.

## Cited references and note


1. G. F. Pfister, *In Search of Clusters*, Prentice Hall, Englewood Cliffs, NJ (1995).
2. N. Bowen, C. Polyzois, and R. D. Regan, "Restart Services for Highly Available Systems," *The 7th IEEE Symposium on Parallel and Distributed Processing* (October 1995), pp. 596–601.
3. J. Nick, J.-Y. Chung, and N. Bowen, "Overview of IBM System/390 Parallel Sysplex—A Commercial Parallel Processing System," *10th International Parallel Processing Symposium* (April 1996), pp. 488–495.
4. J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen, "S/390 Cluster Technology: Parallel Sysplex," *IBM Systems Journal* **36,** No. 2, 172–201 (1997, this issue).
5. L. Spainhower, J. Isenberg, R. Chillarege, and J. Berding, "Design for Fault-Tolerance in System ES/9000 Model 900," *22nd Symposium on Fault-Tolerant Computing* (July 1992), pp. 38–47.
6. S. Calta, J. deVeer, E. Loizides, and R. Strangwayes, "Enterprise Systems Connection (ESCON) Architecture—System Overview," *IBM Journal of Research and Development* **36**, No. 4, 535–552 (1992).
7. R. Cwiakala, J. Haggar, and H. Yudenfriend, "MVS Dynamic Reconfiguration Management," *IBM Journal of Research and Development* **36**, No. 4, 633–646 (1992).
8. M. Swanson and C. Vignola, "MVS/ESA Coupled-Systems Considerations," *IBM Journal of Research and Development* **36**, No. 4, 667–682 (1992).
9. *OS/390 MVS Programming: Sysplex Services Reference*, GC28-1726, IBM Corporation (September 1996); available through IBM branch offices.
10. J. Antognini, "The Automatic Restart Manager in MVS/SP 5.2.2," *Share 85 Proceedings* (August 1995).
11. H. Garcia-Molina, "Election in a Distributed Computing System," *IEEE Transactions on Computers* **31**, No. 1, 48–59 (January 1982).
12. *OS/390 MVS Installation Exits*, GC28-1753, IBM Corporation (March 1996); available through IBM branch offices.
13. *OS/390 MVS Programming: Authorized Assembler Services Guide*, GC28-1763, IBM Corporation (September 1996); available through IBM branch offices.
14. An application may listen by employing the OS/390 service ENFREQ[15] for event type 38.
15. *OS/390 MVS Programming: Authorized Assembler Services Reference, Volume 2*, GC28-1765, IBM Corporation (September 1996); available through IBM branch offices.
16. *MVS/ESA Sysplex Overview*, GC28-1208, IBM Corporation (December 1994); available through IBM branch offices.



*Accepted for publication January 30, 1997.*



**Nicholas S. Bowen** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: bowenn@watson.ibm.com).* Dr. Bowen received the B.S. degree in computer science from the University of Vermont, the M.S. degree in computer engineering from Syracuse University, and the Ph.D. in electrical and computer engineering from the University of Massachusetts at Amherst. He joined IBM at East Fishkill, New York, in 1983 and moved to the Research Center in 1986, where he is currently the Department Group Manager of Servers. He is a senior member of IEEE and a member of ACM. His research interests are operating systems, computer architecture, and fault-tolerant computing.

**James Antognini** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: antogni@watson.ibm.com).* Dr. Antognini is currently a senior programmer at the Research Center, working on projects related to OS/390 and to servers and clients (running on MVS, AIX®, and Windows NT®). He previously worked on PL/I language performance, Enterprise Storage Manager (architecture), integrated records management, asynchronous data mover, automatic restart manager, and the intelligent data miner. He has presented papers at SHARE on MVS, CICS™, and PL/I. Dr. Antognini received the A.B. in psychology from Stanford University in 1970 and the Ph.D. in experimental psychology from Yale University in 1975.

**Richard D. Regan** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: regan@watson.ibm.com).* Mr. Regan received a B.S. in computer science/mathematics from Syracuse University in 1985 and joined IBM that year. He subsequently received an M.S. in computer science from Rensselaer Polytechnic Institute in 1992 and is currently pursuing a Ph.D. in computer science at Polytechnic University. He has been involved in the development of several OS/390 products, including Advanced Peer-to-Peer

Communications (APPC) and the automatic restart manager (ARM). He is currently an advisory software engineer doing research and development of distributed transaction processing systems. Mr. Regan received a Research Division Award in 1995.

**Nicholas C. Matsakis** *IBM S/390 Division, 522 South Road, Poughkeepsie, New York 12601 (electronic mail: nmatsakis @vnet.ibm.com).* Mr. Matsakis is a staff programmer in OS/390 XCF/ARM Development. He graduated in 1984 from Rutgers College with a computer science and economics degree. He has been with IBM for 12 years during which eight of those years have been in the development of the OS/390 cross-system coupling facility (XCF), cross-system extended services (XES), recovery and termination manager (RTM), and automatic restart manager (ARM). He is currently an IBM technical consultant and a member of the OS/390 Parallel Sysplex Enablement Team.

Reprint Order No. G321-5644.