

Failure Recovery for Distributed Processes in Single System Image Clusters

Jeffrey Zabarsky

Tandem Computers, a Compaq Company
390 North Sepulveda Boulevard, Suite 3050
El Segundo, CA 90245

ABSTRACT

Single System Image (SSI) Distributed Operating Systems have been the subject of increasing interest in recent years. This interest has been fueled primarily by the trend towards hardware designs that address the scalability problems of traditional Symmetric Multiprocessor (SMP) architectures. These architectures run the gamut between inexpensive compute nodes connected by high-speed interconnects and architectures in which some or all memory is shared between nodes.

As machines scale to large numbers of nodes, it becomes increasingly intolerable to allow the failure of any one single node to bring down an entire system. Handling failures can dramatically improve the overall system reliability and availability. Amongst the various components of a distributed operating system, the distributed processing component provides significant failure recovery challenges. This is owing to the large number of relationships processes can participate in and the potential for process state to be distributed over many nodes.

This paper presents a failure recovery design for the distributed processing component of Tandem's NonStop Clusters for Unixware SSI distributed computing technology. The failure handling issues, design objectives, detailed design, and implementation experience will be presented. In the end, it will be shown it is possible to construct efficient and robust mechanisms to track relationships and maintain appropriate SSI semantics in the face of arbitrary failures in a clustered environment.

1 Introduction

Symmetric Multiprocessors (SMPs), though the traditional choice for high performance computing, are limited in the number of processors they can support. Models incorporating large numbers of processors can also be very expensive to produce. Large-scale SMP machines can also pose reliability problems in that any hardware or operating system software fault can result in the failure of the entire system. These issues have lead to the exploration of alternative hardware and software designs.

The Intel Paragon [7] is an early example of one potential alternative architecture. The Paragon is a Massively Parallel Processor (MPP) constructed from inexpensive commodity microprocessors connected by a fast interconnect. This system is a representative of the No Remote Memory Access (NORMA) architecture in which each processor has its own local memory and is unable to directly access remote memory. Another architecture design, one that is gaining popularity, involves Non-Uniform Memory Access (NUMA). In this architecture, nodes have access to all memory, but unlike traditional SMPs, incur greater cost accessing remote memory. This architecture is typified by the Stanford FLASH processor [10]. Architectures including both local and global memory are possible with the Scalable Coherent Interface (SCI) [6] representing one possible implementation.

The original Locus TNC distributed operating system extensions work performed for OSF/1 AD on the Intel Paragon [18] as well as the recent work on Solaris MC [8] and Cray UNICOS/mk [1] demonstrate continued interest in distributed operating systems for NORMA environments. The Stanford Hive Project [3] demonstrates that the application of distributed operating system principles can also improve performance and reliability in NUMA environments.

A distributed operating system environment, by its very nature, introduces a structuring methodology that reduces reliance on centralized data structures. This eliminates the typical SMP overhead incurred when processes executing on different processors share common data structures. A Single System Image implementation ensures that programs designed to execute in a traditional SMP environment will continue to run unmodified.

Given a Mean Time to Failure (MTTF) for an individual node, the MTTF for a group of N nodes [5] is given by $MTTF(\text{Group } N) = MTTF(\text{Node})/N$. This clearly shows that, as the system scales to large numbers of nodes, the system's overall MTTF decreases as the inverse of its size. A distributed operating system offers the potential to improve the overall system reliability and availability of highly scaled systems by providing the opportunity for the system to survive the loss of a node.

The distributed process management portion of a distributed operating system presents significant challenges to a failure recovery implementation. In UNIX, processes can participate in a great number of relationships involving, for example, parents/children, process groups, sessions and controlling terminals. These relationships are further complicated by the possibility that the participating processes may be spread among a number of nodes.

Before presenting details of the failure recovery scheme, related research and background information is presented. After the background material, details of the NonStop Clusters for Unixware distributed computing technology are discussed with an emphasis on the distributed process management subsystem. This is followed by a discussion of design motivation, the details of the design, test scenarios, and implementation experience. This paper will show that it is possible to construct efficient and robust mechanisms to track relationships and maintain appropriate SSI semantics in the face of arbitrary failures in a clustered multicomputer environment.

2 Related Research

Besides the LOCUS OS, TCF, TNC, and NonStop Clusters work described later in this section, distributed process management has been pursued in a number of other projects. Below we briefly examine the work performed in this area in the Sprite distributed UNIX system, the Cray UNICOS/mk system, OSF/1 AD on the Intel Paragon, OSF/1 AD Version 2, and Solaris MC.

The Sprite system [4] allows processes to migrate from the workstation on which they are started and later migrate back, but does not support general distributed processing. In addition, because it was architected for networks of desktop systems, the failure model for processes was essentially: if the login node (desktop) died, all processes originating from that node would be killed. If a node on which some processing had migrated died, the login node would simulate the death of processes there. This model, which is appropriate for the environment in which it was designed, presents a fairly simple failure recovery model, but may be inappropriate for a general server multicomputer.

The Cray UNICOS/mk system is based on the Chorus microkernel and does support distributed processing. Process management relationship tracking is performed on a single central node. The failure model (unimplemented per [2]) consists of allowing compute nodes to fail without taking the entire system down. Failure of the process management node results in complete system failure.

The OSF/1 AD [18] system on the Intel Paragon does not provide any support for failure recovery (for distributed processes or any other resources). This system contains an early version of the TNC distributed processing extensions.

The OSF/1 AD Version 2 system provides a centralized process manager node similar to the UNICOS/mk implementation. Similar to Sprite, [12] proposes a simple failure recovery design for OSF/1 AD Version 2 where all processes for which the failed node was an origin are terminated. This provides an extremely simple failure recovery algorithm, but at the expense of terminating processes that would otherwise survive the node failure. This algorithm preserves the cluster following node failures, but sacrifices a potentially large number of processes executing in the cluster.

The Solaris/MC research project, based on the recently published paper [8], does not appear to have addressed process management failure recovery. It appears, though, that the environment intended in that research will have many of the same objectives and deal with many of the same issues as described in this paper.

While this paper does describe UNIX process management failure recovery in a general way, the implementation and experience were derived from providing the capability in Tandem's NonStop Clusters for Unixware environment. NonStop Clusters for Unixware consists of Santa Cruz Operation's industry standard UNIX operating system (Unixware) combined with a set of Tandem kernel extensions designed to allow clustering of multiple computing nodes. The clustering is designed to provide a Single System Image so that all processes, potentially executing on

computing nodes remote from each other, see the same view of the system and are provided all the semantics of a standard UNIX kernel.

The roots of the NonStop Clusters for Unixware technology go back to the early 1980's when the distributed LOCUS OS [15] was a research project at UCLA. The next generation of the technology appeared as the Transparent Computing Facility (TCF) in IBM's AIX operating system for the PS/2 and 370 systems [16]. The LOCUS OS and TCF technologies provided a high level of transparency, but the design of certain algorithms, including failure recovery, limited scalability to clusters of maximum size on the order of 30 nodes. TNC was a new implementation of the technology consisting of a set of separate subsystems. Each subsystem was designed to scale to large clusters and the technology was designed as a set of portable kernel extensions. The distributed processing portion of TNC first appeared in the OSF/1 AD kernel for the Intel Paragon. The complete technology (including such other subsystems as the CFS coherent filesystem, remote device support, and distributed System V IPC has been ported to SVR4.0, SVR4.2 ES/MP, Unixware, and a number of proprietary UNIX platforms.

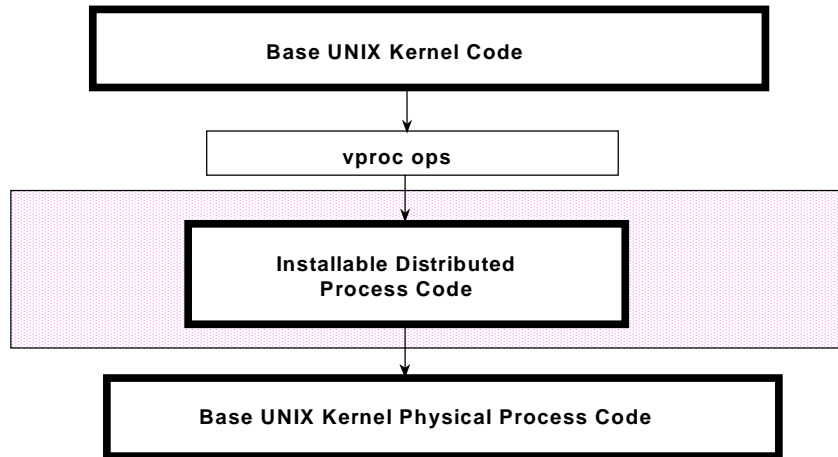
Tandem Computers later acquired the TNC technology and has worked to enhance it to provide high availability. This has been accomplished by eliminating single points of failure that could result in the loss of the entire cluster. Work has included providing filesystem failover (including the root filesystem) and the addition of logic to rebuild subsystem state after node failures. The distributed processing failure handling is one example of the steps taken to ensure cluster availability.

3 Background

A preparatory step towards adding the distributed processing component to the kernel, is the introduction of the Virtual Process (VPROC) abstraction (Figure 1). VPROCs [17, 18] are a technique for structuring process operations in a manner analogous to the filesystem VNODE interface [9]. The VPROC interface consists of a set of Virtual Process Operations (VPOPs), which operate upon a virtual process *vproc* structure (Figure 2). The *vproc* acts as a handle, divorced from the physical process implementation, by which a process may be referenced. Imposing this structure on the process code makes it feasible to implement and install custom process management implementations. The distributed processing code hooks into this interface.

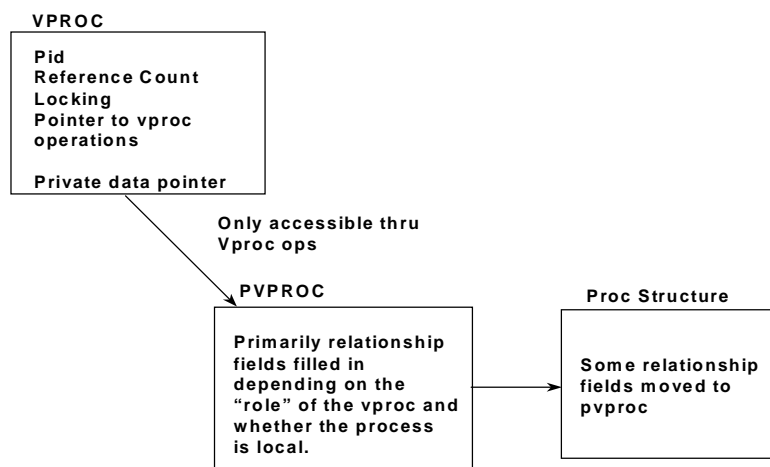
Processes and their *vproc*'s are uniquely identified by a process identifier (*pid*). Distributed processing maintains a single cluster-wide *pid* space by encoding a node number in the upper bits of the *pid*. Processes are assigned *pids* encoded with the number of the node on which they were initially created (known as the *origin node*). At process creation time, a *vproc* is allocated on the origin node and is maintained on that node for the lifetime of the process. This origin node *vproc* acts as a placeholder to prevent reuse of the process's *pid* and plays a role in process tracking when a process moves away from its origin node.

Figure 1: VPROC Restructuring



A process's vproc may exist on nodes in a cluster other than its origin node. A vproc exists at the process's execution node, which may, because of process movement, be different than its origin node. Vprocs are also maintained on the execution nodes of related processes. For example, a parent process's execution node will have vprocs for each of the parent's children. Similarly, each of the children's execution nodes will contain a vproc for the parent. Since vprocs may exist on a particular node to fill more than one role, all vprocs maintain a reference count and only cease to exist when all references are released.

Figure 2: Using VPROCs



UNIX overloads the pid space to uniquely identify both process groups and sessions. Vprocs are similarly overloaded for this purpose. Process group and session lists are maintained in the form of lists of vprocs on the group leader's execution node. Group lists move between nodes with their leaders and may outlive their leaders. The process origin node also serves as the origin for process groups and sessions. Flags in the origin node vproc track the continued existence of process groups and sessions. A subset of the VPOP interfaces is defined to perform process group and session specific operations.

The distributed processing code internally implements the VPOP interfaces in terms of two types of operations: operations to a specific process (PVPOPs) and operations performed to a specific node (PVPSOPs). Both types of operations may result in an RPC to a remote node. Operations on a specific process are performed upon vprocs and result in an RPC if the corresponding process is remote from the calling node. As mentioned above, the execution node of a process is always known by its origin node. To avoid contacting the origin node each time an operation is performed on a remote process, the node on which a remote process was successfully contacted is cached as a hint in the process's vproc on the operation's calling node. If there is no hint for a particular process or the hint is used and found to be stale, the operation is attempted at the process's origin node. If the process is executing at its origin node, the operation succeeds, otherwise the operation fails, but a hint is returned indicating the current execution node. The calling node then retries the operation using the hint and caches the node on success. In the case where the target process is migrating, these steps may have to be retried until the operation catches up to the process.

4 Design Motivation

Distributed operating system components such as filesystems typically have fairly straightforward client-server interactions and lend themselves to simpler forms of failure recovery. In these subsystems, clients care that the server is up and want to return errors if the server fails. Servers, if they keep client state, care about their clients and wish to eliminate state for clients that cease to exist. In contrast, the UNIX process model presents a rich set of potential relationships in a distributed implementation. These relationships include parent/child, process groups and sessions and may consist of a large number of processes spread across a number of nodes. The large number of relationships and the complexity of some of the failure modes pose several challenges to distributed process management failure recovery.

An important requirement for our design was that processes should not be terminated during failure recovery simply because they are related to a failed node. This led to the exploration of three potential solutions for handling node failures:

- Require all caring processes to reliably know where the processes they care about are currently executing.

This approach requires that processes that are going to move (remote exec or migrate) must contact all related processes to inform them of the move. On

failure, the caring processes use this information to determine whether the relationship was severed and cleanup is required. This solution severely impacts the cost of movement by introducing a potentially large amount of additional message traffic to inform caring processes. The tradeoff in this design favors avoiding costs at node failure time in favor of incurring them during process movement.

- Perform global process relationship reconciliation following a node failure.

Global reconciliation requires contacting all nodes and re-gathering process information to determine which relationships have been severed and require cleanup. This approach incurs no run-time expense during normal operation, but requires an extremely message intensive reconciliation following failures. Such a design would also likely require some form of freeze on relationship changes and process movement to ensure generated relationship information does not become stale during reconciliation. This scales poorly to large numbers of nodes in that message traffic will increase non-linearly as nodes are added.

- Track process care relationships such that state is rebuildable and failure recovery can be driven by surviving state.

Tracking occurs dynamically as process relationships are created and destroyed. This approach incurs a slight run-time overhead tracking relationships, but allows efficient scalable recovery without incurring additional expense during process movement.

A fundamental requirement was that the algorithms for performing node failure cleanup must scale to large numbers of nodes. This requirement dictated that the algorithm must be designed such that the required message traffic for recovery should scale linearly with the number of nodes. To do otherwise would result in so called RPC storms during recovery. This decision argued against the global reconciliation approach. This was supported by experience with global reconciliation in LOCUS OS [14] where failure recovery algorithms proved to be a factor in limiting the maximum cluster size to a small number of nodes.

A desirable goal was to keep process movement as inexpensive as possible. Requiring moving processes to contact all related processes would heavily impact the cost of movement. Arguably, process migration may be more useful for purposes of eviction and clean shutdown of cluster nodes rather than for load balancing. If such were the case, extra migration expense would be an acceptable design tradeoff. However, the `rexec()` system call also involves process movement and such extra overhead for remote execution would be detrimental to performance.

These factors argue in favor of the tracking of process care relationships. The following additional requirements motivated the approach taken in the process tracking design:

- Failure recovery must be able to handle arbitrary node failures.

The loss of a single node can effect a large number of process relationships. The failure recovery code must be able to properly handle any and all cleanup required. To provide the greatest assurance that the cluster would remain available in the face of failures, we also required that the algorithm be capable of handling the loss of more than one node at the same time.

- The overall cluster must be available during failure recovery with failures temporarily affecting only those processes related to the failed node(s).

From the outset, distributed processing was designed to avoid bottlenecks caused by centralizing process information on a single node. This was the motivation behind the decentralized origin node process and group tracking strategy outlined above. Similarly [12] suggests decentralizing process management to solve performance problems in OSF/1 AD Version 2. Distributing process management throughout the cluster provides additional benefits for failure recovery in that no one node or group of nodes can result in the loss of all management state.

- Single System Image semantics must be maintained in the face of failures.

Surviving processes should be shielded from all recoverable errors resulting from node failure. Those errors that cannot be recovered from should be mapped to analogous single node error cases. For example, the loss of a child process can be handled by informing the parent that the child was terminated by a signal.

5 Detailed Design of Failure Recovery

The failure recovery design divides into three areas. The first area involves process relationship tracking performed to facilitate failure recovery. Next, relationship information is used to reconstruct origin node state. Finally, the process relationship information is used to drive cleanup of relationships affected by a node loss.

The following assumptions are significant to the failure recovery design:

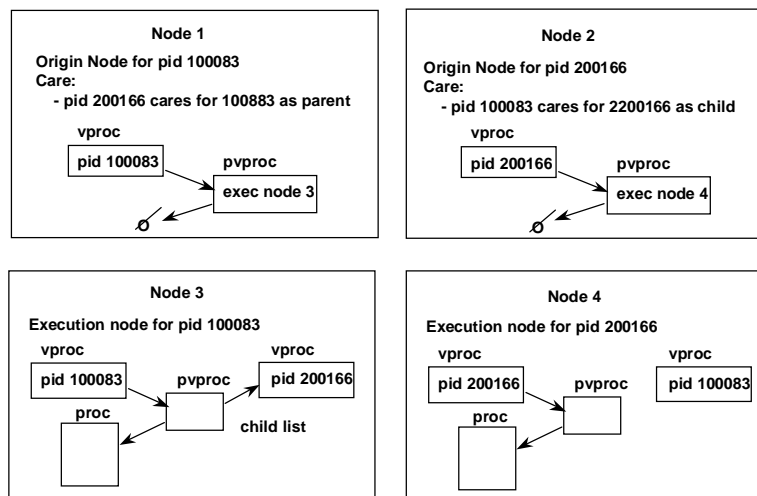
- The cluster has a consistent view of which nodes are currently participating.
- The cluster does not support partitioned or “split brained” operation.
- A node is not allowed to rejoin the cluster until its failure handling has been completed.

5.1 Process Care Relationship Tracking

The goal of tracking care relationships is to replicate state in the cluster in such a way as to allow proper recovery in the event of the failure of any or all nodes participating in a relationship. The approach taken involves tracking all the processes and nodes that care (termed *carers*) about a particular process (termed the *care target*) at the target process’s origin node.

To return to the parent/child example, a parent process cares about its children and a child process cares about its parent. When creating a child process, care relationships are created at both the parent and child origin nodes. The parent origin node tracks a “child cares about parent” care relationship, while the child origin node tracks a “parent cares about child” care relationship. The tracked care relationships are dynamically updated as the relationships themselves are altered. Should the parent exit, in our example, the child would be reassigned the INIT process as its parent. This would result in the destruction of the “child cares about parent” care at the original parent’s origin node and the addition of such a care at the INIT process origin node. When the process finally exits and is reaped, both the “parent cares about child” and “child cares about parent” care relationships are removed.

Figure 3: Sample Care Data for Parent/Child Relationships



Tracking care relationships can incur a potential run-time performance overhead. Steps have been taken to mitigate the costs of tracking by combining (piggybacking) care tracking operations with pre-existing process operations. The PVPSOP_UPDATE_ORIGIN() operation is used to increment/decrement vproc reference counts and set flags at process origin nodes. Whenever possible, origin node care updates are piggybacked on this operation to avoid additional message traffic overhead. In other cases piggybacking is performed where, on average, message traffic would be reduced. An example of this would be the PVPOP_ADD_CHILD_TO_PARENT() operation where the parent’s execution node is contacted. In the likely case in which the parent is executing at its origin, piggybacking the addition of the “child cares about parent” care winds up being beneficial. The resulting care relationships are illustrated in Figure 3.

Care relationship tracking, though so far only mentioned in relation to parents and children, is used to track the following additional relationships:

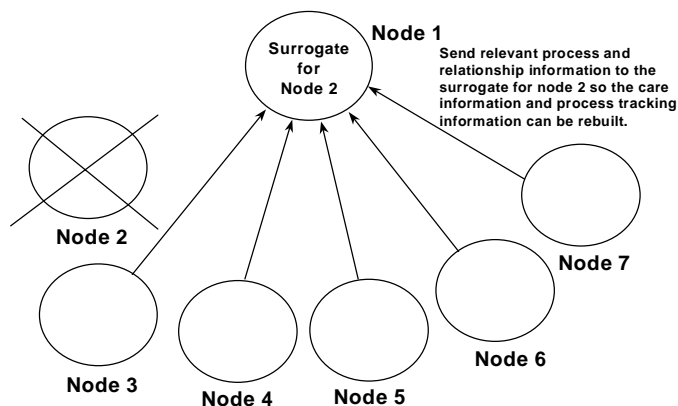
- Foster parent and child (used for accounting).
- Process group leader and members.
- Process session leader and members.
- Controlling terminal node and session leader.
- Controlling terminal node and foreground process group.
- /proc process entry and process

5.2 Recovery from Loss of Origin Node

In addition to any relationships that may be severed, the failure of a node also results in the loss of an origin node for any surviving processes that may have been created on the failed node. Recovery from origin loss involves the setup of what is termed a *surrogate origin node*. This involves selecting a node that will origin for its own processes as well as those belonging to the failed node.

To facilitate having a single node act as origin node for itself and any number of failed nodes, origin related information for a particular node is maintained in a single origin data structure. The structure primarily contains a hash for storing origin node care relationship information. These structures are maintained in an origin node list on each node. In a cluster in which all nodes are participating, each node will have a single entry in its origin node list representing its role as an origin node. Should a node go down, a surrogate node is selected and an additional origin data structure will be added to its list to maintain the failed node's origin information.

Figure 4: Rebuilding Process Tracking Relationship Care Data at the Surrogate



In NonStop Clusters, the Cluster Membership Service (CLMS) is responsible for maintaining state regarding which nodes are in the cluster. The CLMS participates when nodes join the cluster and is responsible for informing remaining nodes when a node has left the cluster. In addition, CLMS provides a facility for selecting and managing surrogate nodes. In the event a node is lost, the CLMS informs the surrogate origin node that it will be surrogating for a node that has just gone down. This results in the creation of an origin data structure marked as being in the pending state. While the surrogate node is marked in a pending state, all operations to the surrogate origin will fail and retry.

After informing the surrogate origin node, the CLMS then informs the cluster that the node has failed. As part of the cleanup triggered by node failure notification, each node generates care relationship information for the surrogate node via two sources. Surviving processes for which the failed node was an origin are examined to create relationship data that should be tracked at the surrogate node. Processes with an origin node other than the failed node are examined for their relationship to processes that do have the failed node as their origin. Special flags are enabled in the care relationship structures to indicate when the target of a care is known to exist on the node generating the care. Each node forwards its list to the designated surrogate origin node (Figure 4) where the cares are added to the pending origin structure's care hash. As the care information is received, the surrogate node uses the care information to recreate origin node vprocs as follows:

- If a “parent cares about child” care is received and it does not contain a flag indicating the child is known to be alive, the vproc is marked as a potential *ghost* process. Ghosts are processes that have been killed by a node failure. They are similar to traditional zombie processes, but must be specially reaped by their parents because they do not have physical process state.
- If a care is received with a flag indicating that the care target is alive, the origin vproc is marked with a flag indicating that the process is still executing and the process location information is set. If a previously received “parent cares about child” care resulted in the vproc being marked with the ghost flag, the flag is disabled.
- If a process group or session care is received with a flag indicating the group leader is still alive, the origin vproc is marked with a flag indicating that the group leader still exists and the leader's location information is set.

As part of the care processing at the surrogate node, appropriate reference counts are placed on the origin vproc. Once all nodes have contacted the surrogate node and the cares have been processed, the surrogate node is marked as active (the pending flag is disabled). At this point the surrogate origin is complete and fully functional and all outstanding origin operations are allowed to complete.

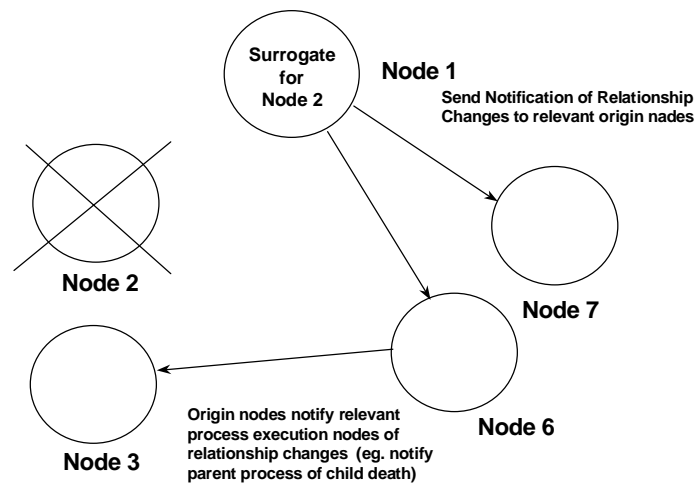
5.3 Cleanup of Severed Care Relationships

As described above, the CLMS notification of a node failure triggers each node to

send care relationship information to the designated origin node. Following this step, each node performs cleanup of care relationships severed by the failure. At the surrogate node, cleanup is performed for all surrogate origin nodes as well as the local origin. The cleanup processing for the failed node is performed at the surrogate origin node after the failed node's surrogate origin is completely setup.

Cleanup is performed by examining all care relationships at an origin. Since care relationships are stored at the target process's origin node, all care relationships can be conveniently examined to determine whether the care target was executing on the failed node. The first step during cleanup is to determine if any process group or session leaders were executing on the failed node. Any lost leaders are reconstructed at the leader's origin node using the "group member cares about leader" care relationship information. Next, care relationships are examined to determine if the care target was lost. Notifications of lost relationships are sent, in the case of care relationships where the carer is a process, to the origin node of the caring process (Figure 5). Care relationships involving node carers are sent to the caring node. Lost relationship notifications to a particular caring process are batched into a single message and sent to the caring process's origin node where cleanup for the relationships is performed.

Figure 5: Surrogate Node Informs Origin Node of Severed Process Relationships



Cleanup of severed relationships is performed by calling the normal distributed process management interfaces. Timing windows exist where care relationship data can be out of sync with the current relationship state. As a result, cleanup code must check for stale care relationships and must be able to handle such cases appropriately.

The following is a list of care relationships and the cleanup steps performed during failure recovery:

- “Parent cares about Child”

In the case where a parent loses a child, the child's origin node notifies the parent's origin node that the child was lost. The parent is informed that the child is in the "ghost" state.
- “Child cares about Parent”

When a child loses its parent, the parent's origin node notifies the child's origin node that the parent was lost. The child is then reassigned INIT as its parent.
- “Foster-Parent cares about Foster-Child”

In the case where a foster parent loses a foster child, the child's origin node notifies the foster parent's origin node that the foster child was lost. The foster child is removed from the foster parent's foster list.
- “Foster-Child cares about Foster-Parent”

When a foster child loses its foster parent, the foster parent's origin node notifies the foster child's origin node that the foster parent was lost. The foster child is then reassigned as a foster child of INIT.
- “Process Group Leader cares about Process Group Member”

When a process group member is lost, the process group member's origin node notifies the process group leader's origin node that the member was lost. The process group member is then removed from the leader's list.
- “Process Session Leader cares about Process Session Member”

Session members are process group leaders, which exist as long as process group members exist. A process group may still exist even if the leader itself is no longer an executing process. A session member (process group leader) is considered lost during a failure if the process group leader and all its members were lost. When a session member is lost, the session member's origin node notifies the session leader's origin node that the member was lost. The session member is removed from the leader's list.
- “Controlling Terminal Node cares about Foreground Process Group”

The origin node of the controlling terminal's foreground process group tracks the care relationship. When the foreground process group is lost, the controlling terminal node is informed and the controlling terminal's foreground process group value is reset.

In addition, while performing cleanup, process groups are examined to determine if they are associated with a controlling terminal as a foreground

process group. A SIGHUP signal is sent to an entire foreground process group if its controlling terminal is lost.

- “/proc Entry Node cares about Process”

Each process has a /proc entry on a fixed node. The "entry cares about process" care relationship is tracked at the process origin. When the process is lost, the process origin node informs the entry node that the process was lost and the entry is cleaned up.

6 Additional Failure Related Design Issues

This section discusses additional design issues that arise when dealing with failures. These include idempotency issues involving process operations, handling races between process movement and nodedown, rejoining the cluster, and recovering from surrogate node loss.

6.1 Process Operation Idempotency

An operation is idempotent or restartable if redoing it an arbitrary number of times is equivalent to doing it once [5]. The original distributed process management implementation, prior to failure handling, had a non-idempotent PVPSOP_REMOTE_VPROC_RELE() operation. This was typically used to decrement the process's origin node vproc hold when the process ceased to exist and was reaped. If the origin node were to fail and the process was in its final stages of going away, the surrogate node may not be created with the process's origin vproc held. Retrying the PVPSOP_REMOTE_VPROC_RELE() operation could therefore result in an extraneous vproc reference count decrement. As a result of this, the PVPSOP_REMOTE_VPROC_RELE() operation was replaced with the PVPSOP_UPDATE_ORIGIN() operation. PVPSOP_UPDATE_ORIGIN() decrements the vproc at the origin node as a side effect of disabling an origin vproc flag. Disabling any or all of the process, process group leader, and/or session group leader origin vproc flags would result in a vproc reference count decrement for each flag disabled. If the specified flag is not enabled in the origin vproc, the vproc reference count is not decremented for that flag. The operation can also be performed to enable flags at the origin (i.e. the process group leader origin flag is enabled when a process becomes a leader) and the origin vproc reference count is incremented for those flags that were not already enabled prior to the operation.

The PVPOP_REASSIGN_CHILD() operation in turn calls the PVPOP_ADD_CHILD_TO_PARENT() operation. A failure could potentially occur after PVPOP_ADD_CHILD_TO_PARENT(), but before the PVPOP_REASSIGN_CHILD() operation could return to its caller. The subsequent retry of the PVPOP_REASSIGN_CHILD() operation could result in the child being added to its new parent's child list twice. Adding a child to a parent twice would result in list corruption. Other similar cases exist where duplicate list additions can

occur. As a result, all list additions must check to make sure the process being added does not already exist on the list.

6.2 Handling Races Between Process Movement and Nodedown

As part of process movement, the origin node of the moving process must be informed of the new execution node. Before failure handling was implemented, it was sufficient to inform the origin node just prior to process movement. If the movement failed, the origin node was contacted again to reverse the change. This simple scheme is inadequate in the face of failures. A failure of the destination node following the origin node update could result in a race between failure recovery and the origin node update to reset the execution node. This race could result in failure recovery incorrectly assuming the process was lost. Alternatively, the origin node could be informed of the new execution node following a successful process movement, but a similar failure recovery race could occur if the original execution node were to fail.

The solution arrived at was to modify process tracking to handle both a to-node and a from-node value instead of a single execution node value. When a process moves, it informs its origin node that it is leaving its from-node and moving to its to-node. During failure recovery, if one of the node values corresponds to the failed node, the other node value must be the current execution node of the process if it survived the failure.

Another potential race can occur when a process's origin node fails while the process is moving. The destination node of the process movement could be examined before the process arrives and the originating node could be examined after the process has left. This race would result in the process's surrogate origin node being constructed without any knowledge of the process, while the process continues to exist.

To catch this race and other races of this type, node failures are assigned a failure sequence number. A moving process carries the failure sequence number from the node it is leaving. This sequence number is compared to the failure sequence number at the destination node. If the destination node's sequence number is higher, a node has failed since beginning the process movement. In this case we return a retry error back to the originating node. In the case where the node originating the process movement survives, the process's origin node knows about the process even if the origin was rebuilt while the process was in flight. The retry error is recognized at the originating node and the movement is re-attempted. In the case where the node originating the process movement is down, the moving process is lost. In either case, we are assured that the process will never survive without the surrogate node's knowledge.

6.3 Rejoining the Cluster

When a failed node attempts to rejoin the cluster it must resume its role as origin node for its processes. The rejoin attempt triggers CLMS to inform the node's

surrogate origin. In the current implementation, the surrogate origin discards all origin information for the node and the same origin failure recovery steps outlined above (sans the final cleanup step) are performed to reconstruct the origin state at the rejoining node. In the future, node rejoin may be made more efficient with the addition of support for pushing origin node information from the surrogate to the rejoining node.

6.4 Recovering from Loss of Surrogate Node

In the event the surrogate node is lost, the CLMS selects a new surrogate node. The surrogate node is reconstructed using the normal node failure protocol (like node rejoin, sans the final cleanup step). The protocol is kicked off for each node that ever participated in the cluster and is currently down.

7 Testing Failure Recovery

The single most useful distributed processing test has proven to be what we have come to term the *foobar test*. The foobar test consists of a simple shell script of the form:

```
while true
do
    <rproc test> &
done
```

Where `<rproc test>` is a test that exercises a particular remote processing system call. The two tests typically substituted for `<rproc test>` are `t_migrate` and `t_rfork`. The `t_migrate` test is a program that calls the `migrate()` system call to migrate itself between a list of nodes. The `t_rfork` command calls `rfork()` to remotely fork processes to a list of nodes.

Because the `<rproc test>` is run as a background task in the shell loop, large numbers of test processes are created. This test puts tremendous stress on the underlying transport mechanisms and exposed several bugs in this area during early system development. The level of stress induced by this test also tends to expose timing windows in the distributed process management code. Running the foobar test and inducing node failures has proven to be a highly useful means of exposing subtle race conditions in the failure recovery implementation. Of particular interest is the scenario where the foobar test is run with the `t_migrate` test migrating back and forth between two nodes and a node failure is induced for one of the two nodes.

More traditional system stress tests in which system resources are run at or near exhaustion have also been deployed. The NonStop Clusters load leveler has been used in conjunction with these stress tests to both dynamically migrate processes and statically place them at exec time. In this way, system stress is induced while ensuring the existence of remote process relationships. Another excellent stress test is the San Diego Zoo multicomputer stress test [13], which creates a test environment involving an extremely active set of remote process relationships. Both

types of stress test can be used to test failure handling by arbitrarily inducing node failures.

Another useful test is to run any of the test scenarios mentioned above, induce a node failure, and have the failed node rejoin the cluster. Repeatedly joining and failing a particular node tends to expose bugs where the failure recovery code fails to setup appropriate vproc reference counts. A final useful test is to perform a system shutdown following a series of successful failure/rejoins. Shutdown forces processes to exit and usually exposes missing vproc reference counts that would normally go undetected while processes continue to execute.

These scenarios have proven to be extremely useful tests of the distributed process management failure recovery implementation. A vast number of subtle variations can be applied to these tests to exercise additional failure modes.

8 Implementation Experience

This section discusses lessons learned during the failure recovery implementation. Lessons include deadlock issues, potential areas for improvement, and common implementation bugs.

The failure recovery algorithm was carefully designed to avoid deadlocks between concurrent processing of node failures. As a result, deadlocks have not been a major concern in the implementation. Of those deadlocks observed while testing, the majority resulted when the surrogate origin node was incorrectly setup to indicate a lost process was still alive. This would cause PVPOP operations on such processes to loop continuously.

Early in the implementation, testing revealed one deadlock that provides an interesting lesson. The surrogate origin read/write list lock was held in shared mode while performing cleanup processing. While cleanup was being performed, the failed node rebooted and attempted to rejoin the cluster. This resulted in an attempt to acquire the origin list lock exclusively to remove the origin entry for the rejoining node. The exclusive locking attempt blocked behind the shared lock held by cleanup. Meanwhile, the cleanup code in turn invoked distributed processing code that indirectly re-acquired the origin list lock in shared mode. Because the system implemented locking fairness, this second shared acquisition of the origin list lock wound up blocking behind the exclusive locking attempt and resulted in a deadlock. This bug serves to highlight the fact that it is impermissible for a single context to recursively acquire read/write locks when locking fairness is implemented. This particular deadlock was solved by introducing an additional origin list deactivate lock, which is held in shared mode across cleanup. The deactivate lock and the original origin list lock must both be held in exclusive mode when removing a node from the origin list.

One common early bug involved inaccurate vproc reference counts following failure recovery. Missing vproc reference counts would result in the vproc being freed prematurely. In the end, the major culprit was missing process group and session

related vproc reference count holds. The test described above in which multiple rejoins are performed followed by a system shutdown was instrumental in flushing out these problems.

A potential area for improvement in the current implementation involves the way process groups and sessions are tracked. An alternate implementation would involve tracking process groups and sessions at the leader's origin instead of the leader's execution node. This change could potentially provide a number of benefits. Moving the group list to the origin node would obviate the need to track group member cares, since this data would be redundant because it would be co-located on the same node as the actual group list. Moving the group lists to the origin node would also greatly simplify the code for reconstructing group lists following failure.

Process tracking related race conditions were a source of a number of bugs. The process tracking code itself is fairly complicated when divorced from failure recovery issues. When combined with failure recovery a number of race conditions are introduced. Especially noteworthy is the potential for the surrogate origin node to spontaneously become the origin for processes as a result of failure recovery.

The distributed process failure recovery code has attained a significant level of maturity. The implementation is now close to two years old. Failure testing of other NonStop Cluster components inevitably exercises process failure recovery on a regular basis and it has proven itself to be quite reliable.

9 Conclusions

There are four problems introduced by distributed state [11]: consistency, crash sensitivity, time and space overheads, and complexity. The distributed process management code is designed to provide consistency, reasonable time and space overheads, and manageable complexity. Where it is most vulnerable is in the area of crash sensitivity where the loss of a single node can result in the loss of the entire cluster.

The failure recovery strategy presented in this paper is designed to address crash sensitivity by adding an additional amount of distributed state in the form of replicated relationship data. As discussed in the section describing failure cleanup, timing windows exist where care relationship information can be inconsistent with actual relationship state. These inconsistencies in process care relationship tracking data are detected and tolerated during failure cleanup. The care relationship tracking code invoked during normal system operations is relatively simple in structure and, in practice, has proven highly reliable. Crash sensitivity, therefore, is not increased by this design. The majority of the complexity is introduced in the recovery code itself, which only executes when a failure has already occurred. As noted previously, additional overhead is incurred at run-time to track process care relationships and steps are taken to piggyback tracking overhead on pre-existing process operations. Space overhead for process care relationships tracking is kept reasonable by the fact that care relationships tracking is distributed amongst the origin nodes so that no one node has to bear the entire burden.

10 Acknowledgments

Special thanks to Bruce Walker who originated the design concepts underlying this paper and worked with me to create an implementable design. I would also like to thank Bruce Walker and Joe Hopfield for their support and encouragement. Thanks to Roman Zajcew, Chris Peak, and Herb Morris for their past contributions to the distributed processing implementation. Thanks also to Laura Ramirez for her efforts to test process failure handling to within an inch of its life. Further thanks to the entire NonStop Clusters team with whom it's been a joy to work.

11 References

1. [Broner 95] Broner, G. UNICOS/mk: A Distributed Operating System with Single System Image, In *Proceedings of the Spring 1995 CUG*.
2. [Broner 96] Broner, G. Personal Communication.
3. [Chapin 95] Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., Gupta, A. Hive: Fault Containment for Shared-Memory Multiprocessors, In *ACM 15th Symposium on Operating Systems Principles*, December, 1995. Available from <http://www-flash.stanford.edu/OS/sosp95-hive/hive.html>
4. [Douglis 90] Douglis, F. Transparent Process Migration in the Sprite Operating System, Thesis, September 1990. Available at <ftp://ftp.cs.berkeley.edu/ucb/sprite/sprite.papers.html>
5. [Gray 93] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.
6. [Gustavson 95] Gustavson, D., Li, Qiang. Local-Area MultiProcessor: the Scalable Coherent Interface. Available at <http://sunrise.scu.edu>
7. [Intel 92] Intel Paragon XP/S Supercomputer Spec Sheet. Intel Corporation, 1992.
8. [Khalidi 96] Khalidi, Y., Bernabeu, J., Matena, V., Shirriff, K., Thadani, M. Solaris MC: A Multi Computer OS, In *Proceedings of the Winter 1996 USENIX Conference*, January 1996. Available at <http://www.sunlabs.com/research/solaris-mc>
9. [Kleiman 86] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the Summer 1986 Usenix Conference*.
10. [Kuskin 94] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessey, J. The Stanford FLASH Processor, In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994. Available at <http://www-flash.stanford.edu/architecture/papers>.
11. [Ousterhout 90] Ousterhout, J. The Role of Distributed Sate. Available from <ftp://ftp.cs.berkeley.edu/ucb/sprite/sprite.papers.html>

12. [Patience 93] Patience, S., Rabii, F. The Design of the Process Management Component of OSF/1 AD Version 2. Available from <http://www.osf.org/os/os.coll.papers/Vol3/index.html>
13. [Peak 93] Peak, C. The San Diego "Zoo": A multicomputer stress test suite. In *Proceedings of the Winter 1993 USENIX Conference*, January 1993.
14. [Popek 85] Popek, G., Walker, B. *The LOCUS Distributed System Architecture*, MIT Press, 1985.
15. [Walker 83] Walker, B., Popek, G., English, B., Kline, C., Thiel, G. The LOCUS Distributed Operating System. In *Proceedings of the Ninth Symposium on Operating System Principles*, October, 1983.
16. [Walker 89] Walker, B., Popek, J. Distributed UNIX Transparency: Goals, Benefits, and the TCF Example. In *Proceedings of the Winter 1989 Uniform Conference*.
17. [Walker 93] Walker, B., Zajcew, R., Thiel, G. Vprocs: A Process Abstraction to Enable Single System Image Distributed Computing. Unpublished White Paper.
18. [Zajcew 93] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*, January 1993. Available at <http://www.osf.org/os/os.coll.papers/Vol2/index.html>

12 Author Information

Jeffrey Zabarsky is a senior developer of NonStop Clusters for Unixware. He has spent the last five years working on various aspects of the Single System Unix clustering technology. He has primarily worked in the area of distributed processing and parallel STREAMs based distributed networking. Additional areas have included distributed filesystem out-of-space handling and distributed /proc. He can be reached at zabby@ladev.tandem.com.