Distributing Process Management in OSF/1 AD Version 3

Faramarz Rabii, Dave Noveck, Stephen Gilbane

OSF Research Institute

rabii@osf.org

dnoveck@osf.org

steveg@osf.org

March 4, 1996

Abstract

In the existing OSF/1 AD Version 2 server, management of a UNIX process has been distributed by dividing it into two parts. Handling of operations involving a single process is done by a per-node server, while handling of operations involving multiple processes and their relationships is done by a single centralized server. The goals of this project are to eliminate the problems that this creates (such as performance degradation, bottleneck, and single point of failure) by distributing the functionality of the centralized server, while maintaining the traditional UNIX semantics of process management.

1.0 Introduction

In the existing OSF/1 AD Version 2 server, management of a UNIX process has been distributed by dividing it into two parts. Handling of operations involving a single process is done by a per-node server, while handling of operations involving multiple processes and their relationships is done by a single centralized server. This division has required the breaking of the traditional UNIX process structure into 2 parts: a **proc** structure, which keeps local state information, and a **pvproc** structure, which keeps information about relationships the process has (parents, children, and process group chains). The **proc** structure is kept on the *Process Management Agent* (PMA) for the node which is locally running the process. The pvproc structures for all processes in an AD cluster are kept on one node, within a server called the *Process Management Master* (PMM).

This design has several unfortunate consequences:

- **Any operation involving process relationships involves the PMM**. This includes basic operations such as *fork*(2), *wait*(2), and *exit*(2), even when both parent and child are on the same PMA. This causes poor performance due to messages being sent between PMA and PMM. One of the goals of this project is to remove the need for any process management messages when all processes involved are local with respect to one another.
- **The PMM is a performance bottleneck**. Since all PMA(s) need to contact the PMM for process creation/termination the load on the PMM will increase in an unbounded fashion.
- **The PMM is a single point of failure for the cluster**.

2.0 Goals of AD3

The goals of this project are to work toward eliminating the problems mentioned in the previous section, while maintaining the traditional UNIX semantics and also achieving the following additional objectives:

- Add the ability to move a process to a new PMA at *exec*(2) instead of *fork*(2) time, in order to avoid duplication of parent virtual memory.
- Allow for a natural way to build a non-distributed version of the operating system which will by-pass all distribution complexity.

3.0 Current Design

The current design of process management is shown in Figures 1 and 2. Each process running in an AD server cluster is represented by (at least) two objects: a **vproc** structure instantiated on the PMA that maintains the process' physical state, and a **vproc** on the PMM.

The **vproc** is a virtualized representation of a process which allows multiple servers in addition to the one servicing system calls for a given process to perform process management actions on a given process.

A **vproc** structure contains a **vdata** structure, which contains two unions representing handles to the local process state and the state of process relationships, respectively, as described below:

- **PROCESS STATE**: On the PMA which is running a given process, the **vdata** structure in that process' **vproc** structure contains a pointer to a **proc** structure. This is the equivalent in AD of the traditional UNIX **proc** structure. On any other server which needs to represent this process (the PMM or a remote PMA sending a signal to this process, for example), the **vdata** in that process' **vproc** contains an **ad_service_t** handle used to contact the PMA running the process.
- **RELATIONSHIP STATE**: On the PMM, the **vdata** contains a pointer to a **pvproc** structure, which contains the links to all the relationship lists to which this process belongs. The

pvproc structure is used to build the traditional UNIX parent-child and session-process-group relationship trees for all processes on all nodes in the entire cluster. On all PMAs, this field is a **ad_service_t** handle used to contact the PMM.

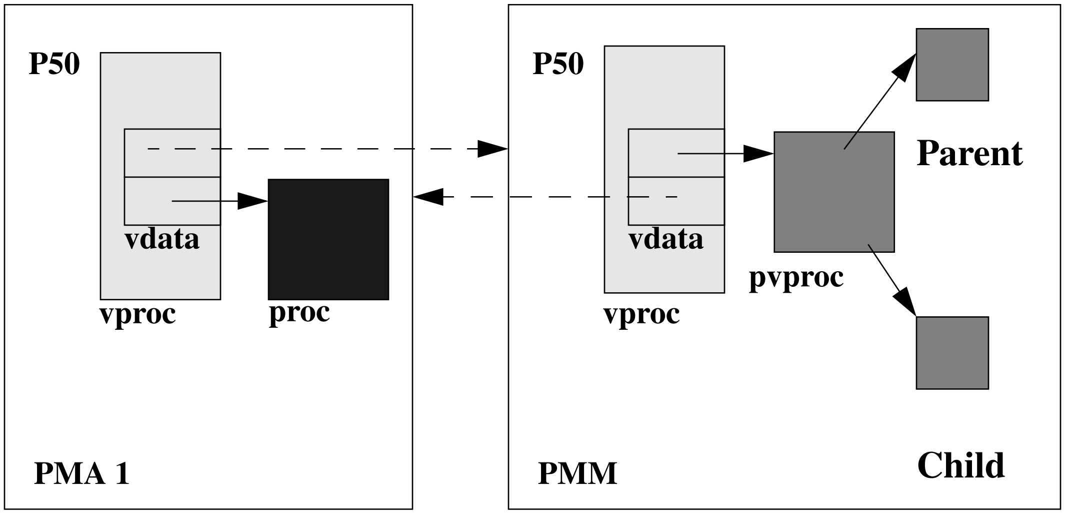

Figure 1. AD2 Design Representing a Process (PID 50)

Flags in the **vproc** structure indicate whether a given process is local or remote to a given server.

A PMA needs to represent a given process under one of two conditions:

- When it creates a local process, and
- When it has to access a remote process (to send it a signal, set the *nice*(2) value, or ptrace a process, for example).

Otherwise, remote processes are not seen or instantiated by PMAs.

The PMM server, on the other hand, maintains a **vproc** for EVERY process in the AD cluster. It creates a relationship tree of all the parent-child relationships in the cluster, with the *init* process (process 1) as the root.

The PMM also creates a **session** structure for each session that is started (which occurs when a process makes a *setsid*(2) system call). A session is made up of a list of process groups, and each process group contains one or more processes. Any UNIX system call that changes parent-child tree (such as *fork(2)* and *exit(2)*) or changes a session or process group tree (such as *setpgrp*(2), *setsid*(2), as well as *fork*(2) and *exit*(2)) must notify the PMM so it can update the appropriate data.

The parent-child communication is bidirectional: when a process exits, both its parent and its child pointers need to be examined and possibly used to perform various operations. Another example of this bidirectional traversal is the determination of orphaned process groups (discussed in detail below). Maintaining the correctness of this bidirectional communication was an important factor in the AD3 design.

4.0 AD3 Design

The overall design objective is to eliminate use of a central process management entity and to distribute its functionality by moving it into the PMA(s), with the overriding goal being improved performance. This goal requires distribution of two main classes of relationships: The **parent/child** relationships and the **session/process group** relationships. General approaches for each are described below, while details are given later in this document.

4.1 Parent-child Relationships

This set of relationships is used to implement *wait*(2) semantics, as well as to determine authorization for certain operations. In practice, a parent at *wait*() time goes through the list of its children in order to determine if a qualified child exists, and if so, certain information from the child is extracted and returned to the calling process. This requires that a process have access to the list of its children. It is also necessary for a process to be able to inform its parent of changes in its state, especially at *exit*() time. Since the child may be running on a different PMA from its parent, there needs be a representation of the child on the parent PMA and vice versa. This is done through the use of the **vproc** abstraction. When a remote process comes to existence, a remote **vproc** for the process will be set up on the parent PMA, which the parent **vproc** will point to. On the child PMA, a remote **vproc** for the parent will be set up to which the child will point.

4.2 Session and Process Group Relationships

Each process in UNIX belongs to a session, uniquely identified in the system by a **sid**, and to a process group, identified by a **pgid**. These entities are used in UNIX to implement job control.

In AD3, sessions and process groups are represented by C++ objects. When a new session or a new process group is created, a **master** object for the session or process group is established on the PMA where the creation occurs. When a process either comes to existence or joins a session or process group on a PMA remote from the master object, a **slave** object corresponding to the session or process group is created to which the process will point. Both master and slave objects for sessions and process groups contain methods which will allow processes to perform any operation that UNIX semantics require.

4.3 The Orphaned Process Group Problem

One problem which is complex enough to warrant special treatment is that of job control. Each process group has a job control eligibility count associated with it, referred to here as **jobc,** which is used to send certain signals to its members, under the conditions described below. The **jobc** count represents the count of processes in a process group which have a parent in the same session but in a different process group. For example, in Figure 2, Process Group 54 has a **jobc** count of 1 because it has 1 process (Process 54) whose parent (Process 51) is in a different process group than itself. Presumably, Process 54 called *setpgrp*(2) at some point to establish a new process group.

The value of **jobc** for a process group is potentially modified when a process leaves or joins a process group, which can happen at fork time, exit time, or when a process calls *setpgrp*(2). When any of the above occurs, calls are made to a routine called *fixjobc*(), which will examine the process' **sid** and **pgid**, and that of all its children. It will then modify **jobc** for each process group as necessary.

A process group with 0 **jobc** is called an **orphaned process group** (in Figure 2, Process Group 51). When a process group becomes orphaned and contains a stopped process, the signals SIGHUP and SIGCONT are posted to all its members, ensuring that they will either exit cleanly or finish to completion, depending on how they have been set to handle these signals.

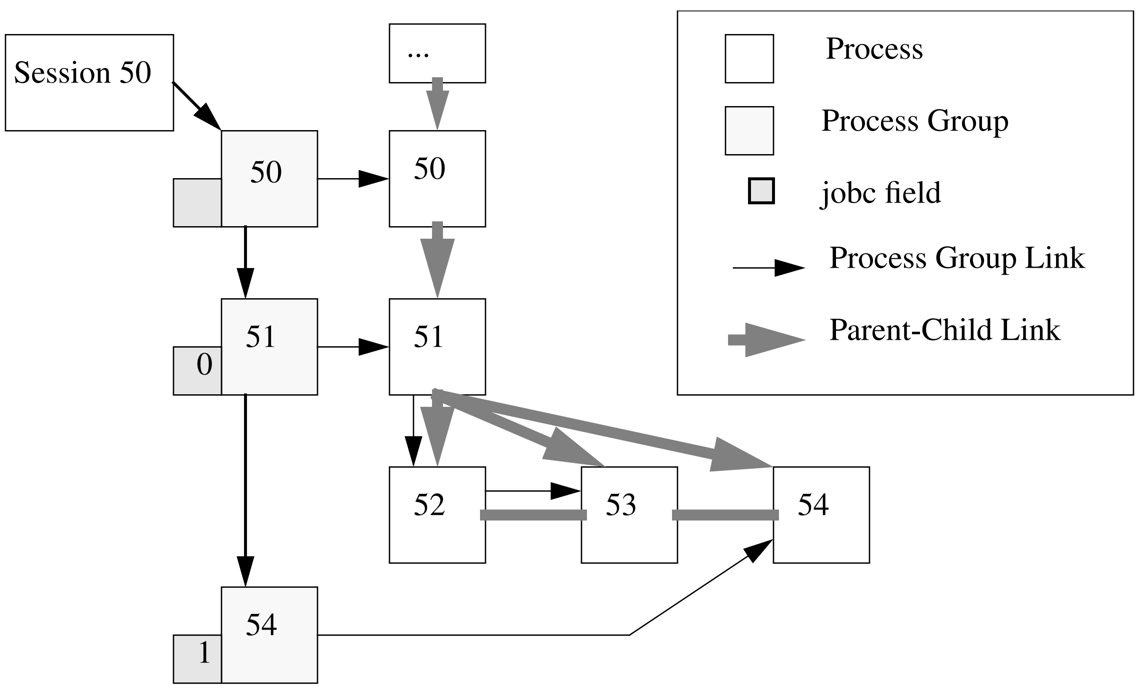

Figure 2. Orphaned Group Problem

5.0 Distributing Parent-child Relationships

As described earlier, AD3, like AD2, will represent each process with a **vproc** structure. The parent-child relationship will be distributed by allowing each process to point to a **vproc** for its parent and a list of its children, with each child process being represented by a **vproc.** When any of the parents or children of a process are remote, a remote **vproc** will be instantiated, to which the

vproc of the process will point and on which it will perform operations. Details of the architecture are given below.

5.1 Summary of Standard UNIX Parent-Child Handling

In non-distributed UNIX, the parent process points to its first child, which in turn points to its siblings. Each child, in turn, points back to its parent. These pointers are updated at several points:

- **When a new child is forked**: the child process is added to the parent list of children. Furthermore, the child's parent pointer is set to point to the parent.
- **When a process exits**: the exiting process's children are reparented to *init* by replacing the parent pointer of each child with a pointer to *init* and adding the child to the list of *init*'s children. The exiting process itself remains in its parent's children list and continues to point to its original parent, while adding SNOTASK to its p_flag.
- **When a process waits on its children**: when a waiting parent discovers a child with the SNOTASK bit set in its **p_flag**. It "reaps" the child, removing the child's proc from the parent's child list and deallocating it.

Note that the parent-child links are only traversed by the parent, while the parent pointer is examined by both the child and the parent.

5.2 AD3 Distribution Architecture

In the AD3 distributed PM, several architectural changes are made:

- **Init**. The *init*(8) process will no longer become the parent of processes whose real parent has exited. This design would put undue pressure on the PMA running *init*, making it a bottleneck and a single point of failure. Instead, when a process's parent exits, the parent will be marked as non-referencible in the normal UNIX (P_REF) way, as described later. Each process will either be reaped by the parent or, if the parent is not available (it has exited or is not interested), the PMA will tear down the process when it exits
- **Process State Tokens**. Each process will have access to the **vproc**'s of both its parent and its children, on which operations may be performed to access locked information about **sid**, **pgid**, and process existence. These operations will be implemented using the AD token module. When this information needs to be obtained, the process will contact the local token client, which, in turn, will result in a cache lookup. If the cache lookup is successful (i.e., a valid token has already been cached), the reference count on the token will be incremented, and the requested information will be returned to the caller. Otherwise, the process will contact the token server on the PMA that is running the target process. The token server will return a token and the information requested. When a process needs to modify its state, it must recall **all** of these outstanding tokens on its state before making any modifications. The above design will give a process all it needs to handle parent-child relationships. This infor-

mation will be set, used, and torn down as described later. An alternate implementation for the single-node configuration will provide the same guarantees using locks alone and will not require use of the token module.

- **Parent vproc**. Access to a process' parent **vproc** via the parent **vproc** pointer will never require a lock. This design decision will simplify the process relationship locking and is feasible because the parent pointer in a **vproc** is never modified after creation. In order to insure the parent pointer will not point to decollated data, a **vproc** (remote or otherwise) will keep a count of the children that are pointing to it. Even when a process has exited and been reaped, if the children count is non-zero, the actual **vproc** is not decollated. This will allow the children to access their parent pointer without requiring the use of a lock to prevent the parent from disappearing.
- **Child vproc**. Access to children will be done under a global lock. In contrast to the parent pointer, child pointers do change, and therefore, access to them must be protected. A single global lock will be used to assure this protection.
- **Wait(2)**. Wait will be done at the parent's PMA. At exit time, a process will go to the parent PMA and attempt to acquire a reference (akin to P_REF) on its parent **vproc**. If this succeeds, it means the parent is still running; the parent now cannot exit until this reference is dropped. The process will then inform its parent of its exit status, and send it the information required for *wait* to complete via an RPC. The parent will then take this information and tear down the child. When the RPC returns, the child will drop the special reference on the parent, allowing the parent to exit, if the parent wishes to. If the child cannot obtain the parent reference, this indicates that the parent is exiting, and it is now up to the server to reap the child. When the server reaps the child, it will, as part of the process, decrement the child count on the parent process.

5.3 Details of Implementation

The details of the distribution of the parent-child relationship are discussed below.

5.3.1 Process Representation

In AD3, a remote process is represented by a virtual object (the **vproc**) on any PMA. There will be two types of **vproc**'s:

- **A Master vproc** will exist on the PMA where the process is running
- **A Remote vproc** will exit on a PMA where an operation is in progress for a process, or where a parent or children of a process exist. Remote **vproc**'s imply no reference on the master **vproc,** and the process may exit while a remote **vproc** for it exists. However, it is possible to perform an operation on a remote **vproc** which will lock the state of the corresponding process and thus keep the process from exiting or modifying certain of its own state information (such as **sid**, **pgid**).

5.3.2 Representation of Process Relationships

When distributing process relationships, we will keep the following structure for each process:

On the PMA of a given process, the process will be represented by a **vproc**, which will contain pointers to the **vproc** of its parent and to the **vproc** of its first child (if any). The child **vproc** will, in turn, point to the **vproc** of its next sibling, thus keeping a full local set of parent/children **vproc**'s for each process.

The **vproc** for a process running on a given PMA will be the master copy, while **vproc**'s for its parent or children may be either local or remote copies, depending on whether the corresponding processes are remote or not. However, each remote **vproc** will be tied to its master **vproc** such that:

- A *GET_STATUS()* operation on a remote **vproc** is available to retrieve the **sid**, **pgid**, and exit status of the target process. This operation will also prevent the target process from changing this information, or from exiting.
- A *REL_STATUS()* operation which allows the target process to modify its **sid** and its **pgid**, and to exit, undoing the above operation.
- Because a given process contains a pointer to its parent **vproc,** this will constitute a child reference on the copy of the parent **vproc** on the PMA local to the process. This will guarantee the parent **vproc** will not be deallocated until that reference drops to zero, even if the parent process exits and has been fully torn down.

The above characteristics allow a process to make the following assumptions about its parent and children:

- There is always a **vproc** for the parent on the PMA local to a process. Therefore, the parent pointer will never have to be modified for the life of a process. A process will be considered orphaned either if it is explicitly told or it finds out when examining the state of its parent.
- Getting a a copy of the parent/child state, as described above, will guarantee that a process has up-to-date and unchanging information about the existence, **sid**, and **pgid** of its parent/child.
- It is up to each process to invalidate information, via a token revocation operation, on its corresponding remote **vproc's** when its **sid** and/or **pgid** are modified, or when it wants to exit.

6.0 Implementation of Sessions and Process Groups

Each process will join the session and the process group of its parent at creation time. A process may change its session or process group by calling *setsid*(2) or *setpgrp*(2), respectively. In AD3, each session and process group is represented by a C++ object. These objects hold two kinds of information:

- Information regarding **attributes** of the session or the process group, such as the controlling TTY, the ID, etc.
- Information regarding **members** of the session or the process group, such as the session ID of the foreground process group or the lists of members of a process group.

In a non-distributed system, the attributes are kept in the corresponding object, while the object holds pointers which make up the relationship trees. In the distributed case, an object must be accessible from PMA(s) other than the PMA where the object was originally created.

In AD3, a **master/slave** model is used to represent sessions and process groups. The **master** object for a session/process group lives on the PMA where it was originally created, while a **slave** object may be instantiated on any remote PMA where one is needed.

The only object a session object needs to know about is the foreground process group. In AD3, the ID of the foreground process group is kept in the master session object. The process group objects, however, need to know about all member processes.

In AD3, each process will be associated with a C++ object which has methods available to implement all operations that may be performed on a process via a process group operation. This object is an instantiation of a **SprocPgrp** object. Each process group object will, in turn, keep a chain of processes local to it. A master process group object will keep track of not only all processes in its process group on its node, but all PMA(s) that have instantiated a slave process group object for that process group. Process group objects keep track of processes PMA(s) where process group members exist via **SprocPgrp** objects. Details of the above are given in sections below and shown in Figure 3.

6.1 Session Objects

Session objects are used to represent sessions on both local and remote nodes. Below are the session classes:

VIRTUAL SESSION OBJECT (**VSess**): This is the class from which other session objects inherit. This is never instantiated.

MASTER SESSION OBJECT (**MasterSess**): This is created on the PMA where the session was originally created.

SLAVE SESSION OBJECT (**SlaveSess**): This is created on PMA(s) where it is subsequently referenced.

Derivation:

MasterSess <-- VSess

SlaveSession <-- Vsess

There is guarenteed to be only one session object per session ID per PMA.

6.1.1 Details of Use

A master session object (**MasterSess**) is instantiated when a process becomes the session leader, that is, when it executes *setsid*(2). The object reserves the process's **pid** slot at the PMA of origin; this ensures the **pid** value does not get re-used until all references to the session are deleted.

A reference on a **MasterSess** object may represent:

- A local process group object within this session, or
- A remote **session** object corresponding to this session object, or
- An explicit reference.

In other words:

Master session ref = Number of process groups on the local node +
Number of corresponding remote session objects +
Explicit refs taken on local node.

A **MasterSess** object is destroyed when its reference count goes to 0.

A slave session object (**SlaveSess**) is instantiated when a process is created on, or moves to, a new PMA as a part of setting up the new process group object. When a new process group object (see **Pgrp** section below) is instantiated, the system will attempt to look up the corresponding **session** object. If one is found, its ref is bumped up by one; otherwise, a new **SlaveSess** object is instantiated, and its ref count is set to one. This act will also involve contacting the **MasterSess** object and incrementing its ref count.

Slave session ref = Number of process groups on this node +
Explicit refs taken on this node.

An **SlaveSess** object is destroyed when its reference count goes to 0. The process of tearing down the slave session will also contact the master session node and decrement the corresponding **MasterSess** object's ref by one.

An **SlaveSess** object may also be instantiated as a result of an explicit call to *sess_find*() with a **sid** that specifies a session which is not on the local node and which does not have a slave session object that has been previously set up. In this case, *sess_find*() will contact the **MasterSess** object (located by decoding the node portion of the **sid**), bump its ref count, and instantiate a **SlaveSess** object with a ref of 1.

6.2 The Process Group Objects

The **Pgrp** object represents a set of processes within a process group (see *setpgrp*(2)). The **vproc** structure for every processes point to a **Pgrp** object, which defines its process group.

VIRTUAL PGRP OBJECT (**VPgrp**): This is the class from which other **Pgrp** objects are derived. This class is never instantiated.

MASTER PGRP OBJECT (**MasterPgrp**): This object is created on the PMA where the process group was initially created.

SLAVE PGRP OBJECT (**SlavePgrp**): This object is created on remote PMA(s) where other members of a process group exist.

Derivation:

MasterPgrp <-- VPgrp
SlavePgrp <-- VPgrp

There will be at most one **Pgrp** object per process group id per node. There will always be a **MasterPgrp** object on the node where the process group was created. From this object hangs a list of **SprocPgrp** objects (see **SprocPgrp** section below).

When a process is created, a special lookup is done, keyed on the parent **pgid**. This lookup will either find the associated **MasterPgrp** object (which it must do so if parent and child are on the same node), or will create a **SlavePgrp** object and tie it to the **MasterPgrp** via a **SprocPgrp** object, as described above.

6.2.1 Pgrp Data Fields

For details of the fields and methods of **VPgrp**, **MasterPgrp**, and **SlavePgrp** objects, see Appendix A.

6.2.2 Details of Use

A **MasterPgrp** object is instantiated when a process becomes a process group leader. This can only happen as a side effect of the creation of a new session, when a process executes the *setsid*() system call. The act of becoming a session leader creates:

- a **session** object, with its **sid** set to the **pid** of the leader process, and
- a **MasterPgrp** object, with the **pgid** of the session leader.

The **MasterPgrp** object thus established will contain one process in its list of members and will have a reference count of 1.

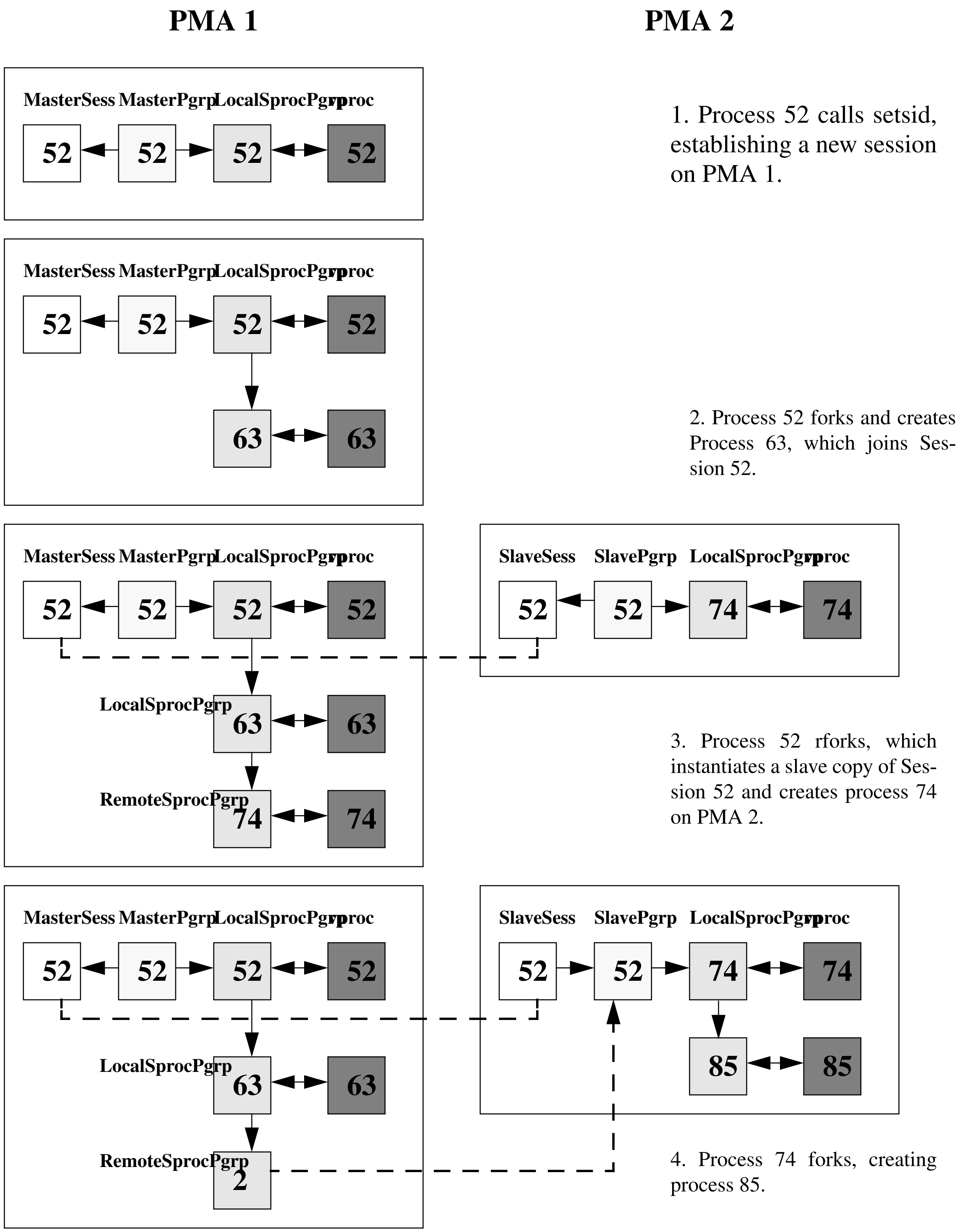

Figure 3. Session/Process Group Design on AD3

A **MasterPgrp** object contains a list of **SprocPgrp** objects, each of which reflect its member processes. The **SprocPgrp** object (described in detail in the next section) comes in two flavors: One which represents a local process in a one-to-one basis, and another which represents a remote PMA with processes which are members of this master process group.

When an additional process joins a process group on its local PMA, it will be inserted in its members list as an **SprocPgrp** object. When a remote PMA becomes the home to one or more processes which are members of a master process group, one **SprocPgrp** object per remote PMA is added to the members list of the process group. Corresponding to the **SprocPgrp** object, **SlavePgrp** object will also be created on the remote PMA.

A reference on a **MasterPgrp** object may represent:

- A local process in this process group, or
- A **SlavePgrp** object corresponding to this object, or
- An explicit reference on this object.

In other words,

MasterPgrp ref =	Number of local processes +
	Number of slave Pgrp objects +
	Number of explicit refs

An **SlavePgrp** is instantiated when a process is created or moves to a PMA that is remote from the MasterPgrp object of its process group. As described above, this entails adding an **SprocPgrp** object to the member list of the **MasterPgrp** object and the creation of a **SlavePgrp** object on the remote PMA. A reference on a **SlavePgrp** object may represent:

- A local processes in this process group, or
- An explicit references on this object.

In other words,

SlavePgrp ref =	Number of local processes +
	Number of explicit refs

6.3 The SprocPgrp Object

A **SprocPgrp** object represents a set of processes on either a local or remote node. These objects linked together in a list form a process group tree, with a **Pgrp** object as the list head. The **SprocPgrp** object allows for a limited set of methods that can be applied to processes within a process group. There are two types of instantiations of this object:

- One which represent **ONE** process in a Pgrp running on the local node; and
- One which represents **ALL** processes in a given process group on a given remote node

The **VSprocPgrp** class is a virtual base class for local (**LocalSprocPgrp**) and remote (**Remote-SprocPgrp**) derived objects, and not directly instantiated:

Derivation:

LocalSprocPgrp <-- VSprocPgrp
RemoteSprocPgrp <-- VSprocPgrp

6.3.1 Data Fields

For details of the **SprocPgrp** fields and methods, see Appendix A.

6.3.2 Details of Use

An **SprocPgrp** object is established under one of two conditions:

1. **Local** (**LocalSprocPgrp**). This object is established when a process is created. It is linked to a **Pgrp** object at the node where the process exists. This association causes the ref of the **Pgrp** to be incremented by one. It is deleted when the process exits, at which time the reference count on the **Pgrp** is decremented. Note that the **Pgrp** may be a master or slave object, depending on whether or not the process resides on the node where the **Pgrp** was originally created.

2. **Remote** (**RemoteSprocPgrp**). This object is established when a **SlavePgrp** is created. This object represents a **SlavePgrp** on the node where the **MasterPgrp** resides. Its establishment will cause the ref on the **MasterPgrp** to be incremented by one. It is deleted when the ref count on the **SlavePgrp** associated with it drops to 0. The deletion of an **SprocPgrp** will cause the ref on **MasterPgrp** to drop by one. Such objects are linked only to the **MasterPgrp**.

The reason we have two versions of the **SprocPgrp** object is that the set of operations on these objects is a common subset of those on processes and those on process groups. Such operations are always initiated as a result of an equivalent operation on a process group. This action will, in turn, result in walking down the chain of **SprocPgrp** objects, invoking the corresponding operation on each **SprocPgrp**.

7.0 The Orphaned Process Group Problem

The handling of orphaned process groups in UNIX ties process group and session relations to those of parent-child. In the AD3 distributed case, an efficient implementation of this requires cached access to the parent and child **pgid** and **sid** for each process. Correctness also requires that the act of **jobc** adjustment be atomic with respect to exiting or switching **pgid** or **sid**.

In the AD3 design, each process has access to either the master **vproc** or to remote **vproc**'s for its parent and children. In the case of remote **vproc**'s, the **pgid/sid** values, as well as whether the corresponding processes have gone through **jobc** adjustment at exit time, is cached. The adjustment of **jobc** for a process group is based on three items:

- The **pgid/sid** of process group being examined
- The **pgid/sid** of the parent process, and
- Whether the parent process has already gone through exit processing of **jobc** adjustment.

The above values are held in the real **vproc** and are accessible from remote **vproc**'s via a token mechanism.

A child or parent process can prevent the update to a process's (remote) **vproc** by referencing the state token on the remote **vproc**, thus blocking the update while it is examining the values and taking action.

When a process wants to switch its **pgrp/sid**, it will reference the status token on the parent (remote) **vproc**, thus preventing changes to the parent state. It will then revoke the status token on its own real and remote **vproc**'s thus preventing anyone from looking at its state until the process is done. The process will then go through its own **Pgrp** and determine if **jobc** adjustment needs to be done. This is done by first determining if the parent has already done so in the process of exiting. If not, the process compares its own **sid/pgid** with that of its parent. The process will then go through the list of its children and decide if each child's **Pgrp jobc** value needs to be updated.

At this point, the children cannot modify their **sid/pgid**, as they need their parent **vproc** status token (held by the current process) to do so. However, the process needs to examine if a child has already exited and thus done the **jobc** adjustment on itself. Therefore, it needs to examine the child state, which is done by taking a reference on the status token of the child (remote) **vproc**. The child state is examined, after which the child (remote) **vproc** token in unreferenced. Since the token reference ordering is parent first, then current process, then children, there is no possibility of deadlock.

The process will then update its **sid/pgid** locally and go through the **jobc** adjustment for the new **Pgrp**, if it has joined a new one, or go on to the next step below if it is exiting.

As the last step, the process will then update the status in its own real **vproc** with the new **sid/pgid** and whether it just went through **jobc** adjustment for exit. Next, it will allow references on its own status token. Finally, the process will drop the reference on the status token of the parent (remote) **vproc**.

8.0 The Uid-Gid Problem

There are certain times when an action must be taken that involves all processes with a given **uid** or **gid**. These include the System V system call *sigsendset*(2) with the idtype set as **P_UID** or **P_GID**. In these cases, AD3 will use the following design:

When a process is created, its PMA will add it to a list of all processes on that node which are members of its **uid** and to another list by its **gid**. If no such list exists, one will be created and a master server will be told to add the node to master lists for that **uid** and **gid**.

When a process exits or switches **uid/gid**, it will be removed from the per-node list, and if the list becomes empty, it will be deleted and the list item representing that node will be removed from the master list.

When an action is taken that involves all members of a **uid/gid**, such as in the above system calls, the master list is consulted and the action is sent to each member node. Each member node will then execute the action on all member processes.

9.0 Migration and Remote Exec

This problem entails moving an existing (running) process from a node of origin to a target node. There are several motivations for allowing this.

REMOTE EXEC. The *rfork()/exec()* paradigm is an inefficient way to create remote processes, since the newly-formed process has its VM region, or at least a part thereof, copied from its parent at *rfork*() time, only to be discarded during the subsequent *exec*(). This problem can be addressed by combining *rfork*() and *exec*() into a single system call (we'll call here *rspawn*(), a remote version of the POSIX *spawn*() system call). This system call would create the remote process and overlay it with the new image all at once. However, many existing UNIX programs rely on the *fork()/exec()* combination, and for them to continue to work efficiently, it is desirable to perform the *fork*() locally and for the policy placement module to migrate the process at *exec*() time to a new node.

MIGRATION LOAD BALANCING. It may also be desirable to be able to relieve the load on an overloaded node by migrating processes off of it. In practice, if jobs are long-lived, the load on the system is predictable, and jobs can be divided among nodes at creation time. On the other hand, if jobs are short lived, then when a node becomes overloaded, it is sufficient to simply create new processes on other nodes and let the short lived jobs on the overloaded node finish, thus reducing the load.

MIGRATION SHUTDOWN. If a node is being shut down, it is desirable to move all its processes to another node.

In our design, we will only address the first issue, which is that of *rexec*().

Remote exec can be implemented similarly to remote fork, which is already present in AD2, with additions to handle a few extra complexities. When a process is co-resident with its parent and/or children, the latter holds pointers to the process' real **vproc**. When a process moves, its parent and/or children need to change this pointer to point to a shadow **vproc**, which must be created to replace the real **vproc**. Also, the token server for the process must move from the node of origin to the target node.

Access to the process must be suspended for the duration of the move. A method must be established for locating the moved process. The **pid** for the moved process must be reserved for the life of the process, and it must be freed when the process exits. Below we discuss two main problems, that of locating the process and that of moving the token server.

9.1 Process Location

There are two situations in which a remote process must be located. One is when a pid needs to be converted to a **vproc,** such as in performing a *kill*(). The other when a parent or child process needs to contact it.

1) In the first case, a process is located by mapping its pid to an **ad_service_t**, which holds the process' location. This is done via the pid broker, which assigns a whole pid range to a given node during start-up. When a pid moves to another node, the above mapping is no longer valid.

The solution we use here is that when a VPOP operation is done on a process which has moved, the operation will go to the node of origin. That node will recognize that the process has moved or begun to move, and will return an error indicating the process has moved, along with the new location. The caller will then retry, using the new node as the target. On the new node, if the process is not done moving yet, the caller will be suspended until the move is complete. Any subsequent moves will require the moving process to inform the node of origin, which will thus act as an information repository for the process' location.

2) In the second case, the parent or child attempts to do a token operation on a remote process. It will first go to the node of origin, which will return both an error, indicating that the process has moved, and a value representing the new location. The caller will then retry (as in the first case) the operation at the new target node. If the move is still in progress, any attempts to perform operations on it will block until the move is complete.

Both these cases are similar enough to have the same implementation.

The above work will also require a new set of interfaces to the token server that would enable a token server object to be moved from one node to another. This operation would pack the client information for a given object and return it to the caller, thus migrating the object from a source to a target token server. Another interface will be required on the target node used to migrate a server object to the new token server. The process management module will block any attempts to contact the two token servers on behalf of the moving object for the duration of move.

10.0 Examples of Remote Operations

Below are examples of some common and important operations.

10.1 Remote fork

Remote fork will involve the following steps.

- The parent will execute the *rfork* system call.
- On the parent PMA, the portion of the parent state needed to create a child is packaged into a data structure. This state includes: handles to open files, credentials, limits, signal related information, parent task port, etc. This information is then sent to the target PMA via an RPC.
- On the target PMA, the RPC sent by parent is received and picked up by a server thread. This thread will proceed to create the child as follows:
- A **pid** is generated for the child. Next, appropriate data structures for a process are created and the information form the parent are installed into the child structures.
- A **vproc** is allocated for the child.
- A lookup is done for the parent **vproc**. If one is found, its child ref count is incremented. If one is not found, then a remote **vproc** for the parent is allocated, and initialized. The child **vproc** is made to point to it via the parent pointer. This sets up the child-parent relationship on the child PMA.
- The parent **Pgrp** object is looked up. If it is found, then it is referenced; if not, one is allocated. This allocation will result in lookup of the parent's **session** object. Again, if it is found, it is referenced; if not, a **session** object is allocated and associated with the master **session** object. The **Pgrp** object is then made to point to the **session** object, and associated with the **MasterPgrp** via a **SprocPgrp** object. Finally, the **vproc** of child is made to point to the **Pgrp** object. This will set up the child process group-session identity.
- The child is set to run
- The RPC will return a success status to the parent node.
- On the parent node, the *rfork* system call will resume. It will allocate a **vproc** for the child and queue it in the list of children of the parent process. This **vproc** will have a reference of 1, indicating the reference its parent has on it.
- The *rfork* system call will return.

10.2 Exit

Exit of a process will involve the following steps.

- The process will call the *exit* system call
- On the process PMA, the system waits until all local refs on the processes **vproc** have

dropped to zero. Then, the exit flag is set in the process, indicating the process can no longer be referenced. The exit path is now single-threaded.

- If the process is a session leader, then SIGHUP is sent to all members of foreground process group. A handle to the controlling TTY, if any, is also extracted from the **session** object and the session object's controlling TTY handle is set to NULL.
- The process will then go through its children list and orphan them. Any child being orphaned which has already exited will be reaped at this time.
- The process will call *clearalias()* on the controlling TTY vnode, if one exists.
- The process will release its local resources. This involves freeing various data structures, closing files, and gathering statistics to be returned to the parent.
- The exit statistics are packaged in a data structure and sent to the parent PMA via a RPC.
- On the parent PMA, a server thread will pick up the RPC. It will attempt to obtain a referenced **vproc** for the parent. If one is found, then the exit information is inserted in a data structure associated with that of the child, and the RPC returns. The parent is then informed that a child has exited. If the parent **vproc** cannot be referenced, it means the parent is already exiting, and the service thread will discard the exit information and deallocate the child resources on the parent PMA. Deallocation of the child **vproc** in any case will cause the child ref on the parent PMA to drop by one.
- The RPC will return to the PMA of the exiting process. At this point, the remaining resources of the exiting process are deallocated. These will include the **vproc** whose deallocation will drop the child ref count on the parent **vproc** by one. If the process had migrated, then an RPC is made to the original PMA, telling it to deallocate the **vproc** on that PMA and to free the **pid** for re-use.
- The child process is then terminated.

10.3 Wait

Waiting for status change in a child is done as follow:

- The parent process will execute *wait* locally.
- The parent process will go through its child list looking for a child whose status matches the wait requirements.
- If an acceptable child is found, then the parent will obtain the required information about it. If the wait was for an exited child, then the child is reaped.
- If no acceptable child is found the parent will sleep, pending notification of a pending child.

10.4 Remote exec

The remote *exec* system call will involve moving an already existing process to a new PMA, while overlaying its address space with a new image. The general case is done as follows:

- A process will execute the *rexec* system call.
- Since *rexec*, like *exec* will overlay the executable image, the executable may not be having multiple threads; therefore, the system will insure the single thread nature of the process.
- The **vproc** of the migrating process is marked with the migrating status and the new target PMA.
- The information regarding a process which remains across a *rexec* is packaged in a data structure. A call is made to the token server for the **vproc** of the migration process. The server information is extracted out, and packaged in the mentioned data structure.
- An RPC is made to the target PMA, with the above information as well as the path to the new image.
- On the target PMA, the server thread will pick up the RPC. The thread will then allocate needed resources for the migrating process, in a manner similar to *rfork* but without generating a new **pid**.
- The **vproc** for the migrating process is inserted into new PMA's hash queues and marked as 'migration in progress'.
- The token server information sent from the PMA or origin is installed in the local token module and associated with the new **vproc**.
- As in *rfork*, a new process is established; however, the task port used for the generation of the underlying mach task is that of the local PMA.
- The normal *exce* work is done to setup the image of the new process.
- The process is made to run.
- The **vproc** of the now migrated process is cleared of the 'migration in progress' bit.
- The RPC will return the target PMA indicating success.

11.0 Implementation approach

The above design will be implemented incrementally by two individuals: Steve Gilbane and Faramarz Rabii. The work will be divided into three main sections as described below.

11.1 Distribution of parent child relationships.

This is the first portion of the work to be implemented. Initially the framework for processes to keep track of their parents and children will be added to the existing PMA code. It will be confirmed that the frame is built and torn down as processes are created and exit. This will result in the work for fork and exit being handled on PMA(s). Once the frame work is in place, the PMM involved in handling "wait" will be moved to the PMA(s), completing the above task.In order to keep the system functional, the session/process group handling will still be done at the PMM and distributed in the next section. Since job control involves interaction between parent/child relationships and that of sessions and process groups, handling of job control will be disabled until next section is done.

11.2 Distribution of session/process groups

Sessions and process groups will be represented as C++ objects to facilitate distribution. This effort will involve three main parts. First, re-writing the code involved in C++ in a non distributed fashion as a part of the single PMM. And second adding the distribution code and moving the objects to PMA(s). This effort is expected the be more complex than distribution of parent child relationships. Finally, once all the support exists, job control will be re-enabled.

11.3 Support for rexec.

This section of the work is independent of the previous two, and will be implemented after the first are done. Moving a process to another PMA requires moving the token state as well as the process, therefore modifications are needed to the token module to allow migration of token server state from one PMA to another. Furthermore, code for rexec and retries for VPOP operations will have to be implemented. Finally the pid allocation and process tracking scheme in the AD system will have to be re-implemented to allow for the migration. This will involve removal of a static process table and its replacement with a hash queue based mechanism.