



Scaling Out OpenSSI

OpenSSI.org

Bruce J. Walker
Jaideep Dharap

HP Labs

Agenda

Introduction to SSI Clusters and OpenSSI

How OpenSSI clusters meet the cluster requirements

Scalability Issues and Strategies

OpenSSI Status

What is a Full Single System Image Solution?

Complete Cluster looks like a single system to:

- Users;
- Administrators;
- Programs;

Co-operating OS Kernels providing transparent access to all OS resources cluster-wide, using a single namespace

- A.K.A – You don't really know it's a cluster!



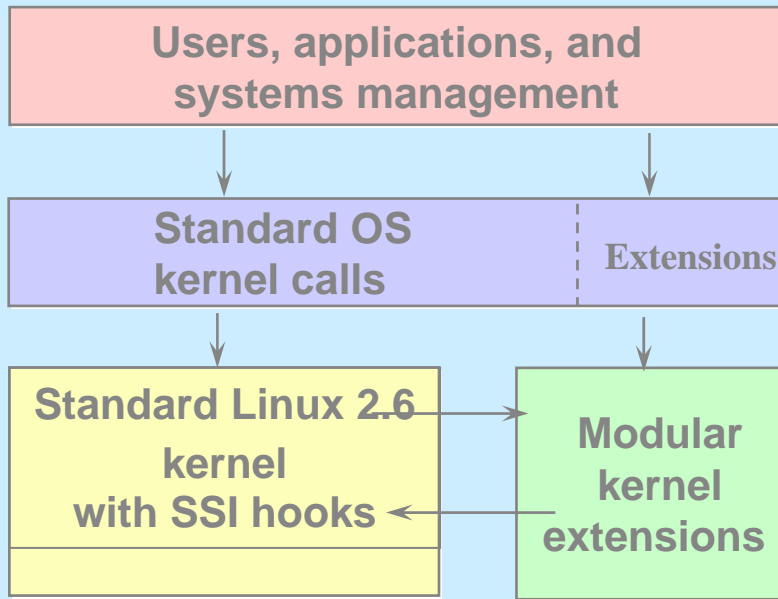
**The state of
cluster nirvana**

Overview of OpenSSI Clusters

- **Single HA root filesystem accessed from all nodes via cluster filesystem**
 - therefore only one Linux install per cluster
- **Instance of Linux Kernel on each node**
 - Working together to provide a Single System Image
- **Single view of filesystems, devices, processes, ipc objects**
 - No cross-node virtual memory;
- **HA of cluster, applications, filesystems and network**
- **Single management domain**
- **Load balancing of connections and processes**

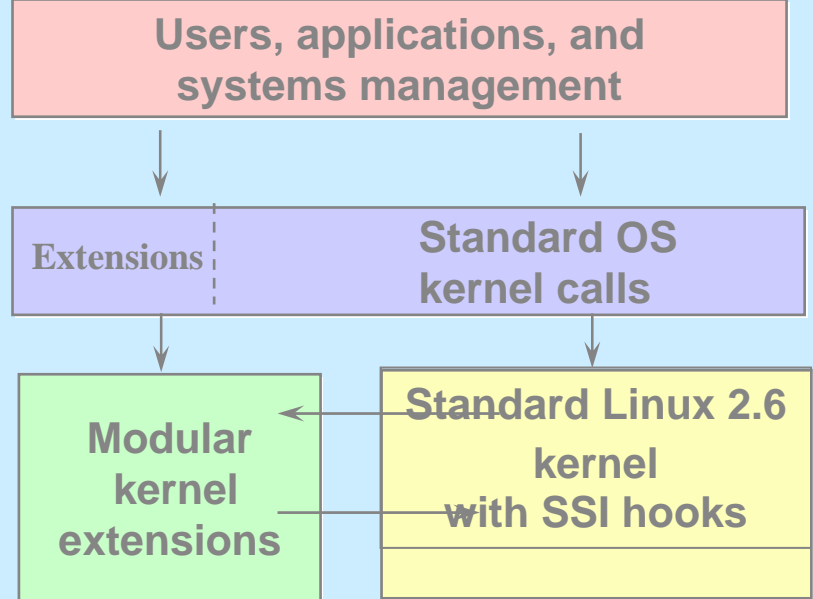
How Does OpenSSI Clustering Work?

Uniprocessor or SMP node



Devices

Uniprocessor or SMP node

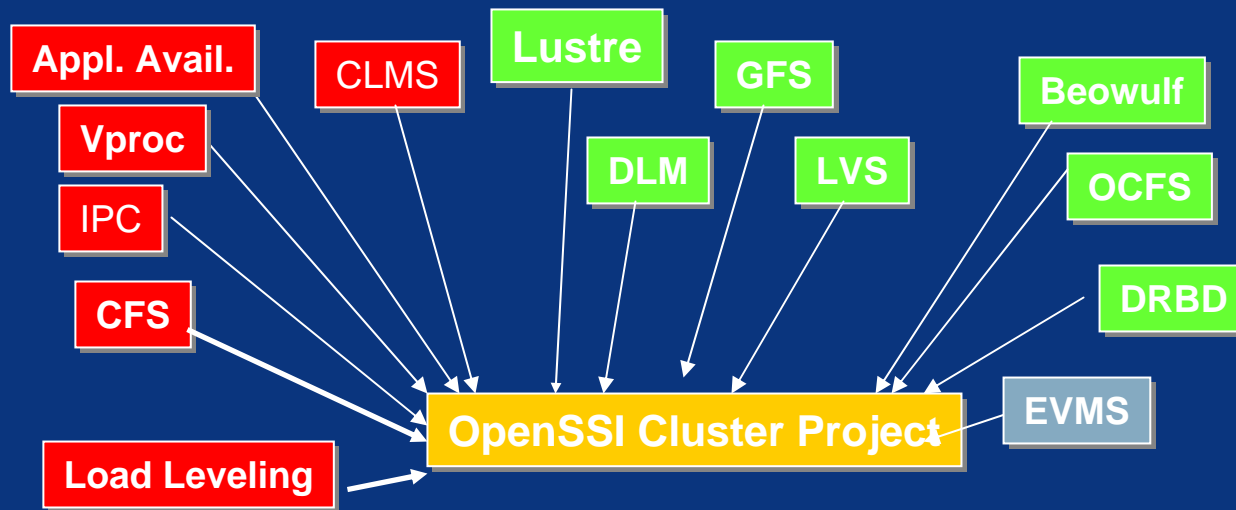


Devices



Other nodes

Component Contributions to OpenSSI Cluster Project



HP contributed

Open source and integrated

To be integrated

OpenSSI Cluster Architecture/ Components

13. Packaging and Install

14. Init; booting;
run levels

15. Sysadmin;

18. Timesync

16. Appl Availability;
HA daemons

17. Application Service
Provisioning

19. MPI, etc.

Kernel
Interface

1. Membership

3. Filesystem

CFS

GFS

5. Process
Loadleveling

6. IPC

Physical
filesystems

Lustre

4. Process Mgmt

7. Networking/
LVS

8. DLM

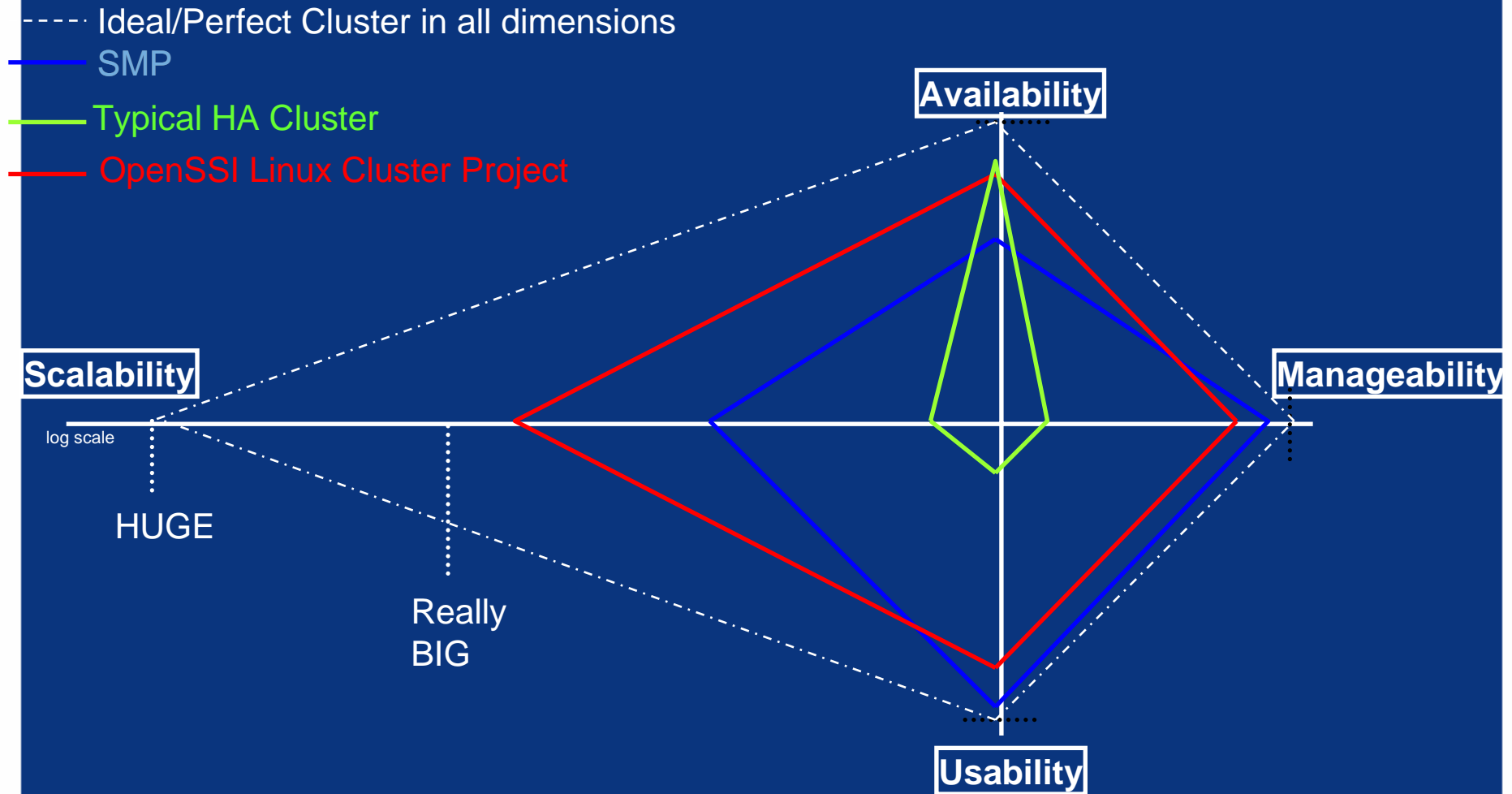
9. Devices/
shared storage
devfs

11. EVMS (TBD)

12. DRBD

2. Internode Communication/
HA interconnect

OpenSSI Linux Cluster



Availability

- **No Single (or even multiple) Point(s) of Failure**
- **Automatic Failover/restart of services in the event of hardware or software failure**
- **Filesystem failover integrated and automatic**
- **Application Availability is simpler in an SSI Cluster environment; statefull restart easily done;**
- **OpenSSI Cluster provides a simpler operator and programming environment**
- **Architected to avoid scheduled downtime**
- **System-level checkpoint restart**
- **Quorum, Stonith and Split-brain avoidance**
- **HA-LVS integrated**

OpenSSI Linux Clusters - Manageability

- **Single Installation**
- **Joining the cluster is automatic as part of booting and doesn't have to be managed**
- **Trivial online addition of new nodes**
- **Use standard single node tools (SSI Admin)**
- **Visibility of all resources of all nodes from any node**
- **Applications, utilities, programmers, users and administrators often needn't be aware of the SSI Cluster**
- **Simpler HA (high availability) management**
- **OS upgrade without application interruption (ongoing)**

OpenSSI Linux Cluster

Ease of Use

- **Can run anything anywhere with no setup;**
- **Can see everything from any node;**
- **service failover/restart is trivial;**
- **automatic or manual load balancing;**
- **powerful environment for application service provisioning, monitoring and re-arranging as needed**

Price / Performance Scalability

What is Scalability?

Environmental Scalability and Application Scalability!

Environmental (Cluster) Scalability:

- more USEABLE processors, memory, I/O, etc.
- SSI makes these added resources useable

Price / Performance Scalability - Application Scalability

- **SSI makes distributing function very easy**
- **SSI allows sharing of resources between processes on different nodes**
- **SSI allows replicated instances to co-ordinate (almost as easy as replicated instances on an SMP; in some ways much better)**
- **Monolithic applications don't "just" scale**
 - **Load balancing of connections and processes**
 - **Selective load balancing**

OpenSSI Clusters

Price/Performance Scalability

- **SSI allows any process on any processor**
 - **general load leveling and incremental growth**
- **All resources transparently visible from all nodes:**
 - **filesystems, IPC, processes, devices*, networking***
- **OS version in local memory on each node**
- **Migrated processes use local resources and not home-node resources**
- **Industry Standard Hardware (can mix hardware)**
- **OS to OS messages minimized**
- **Distributed OS algorithms written to scale to hundreds of nodes (and successful demonstrated to 133 blades and 27 Itanium SMP nodes)**

What about Scaling much higher

Potential Problem Areas:

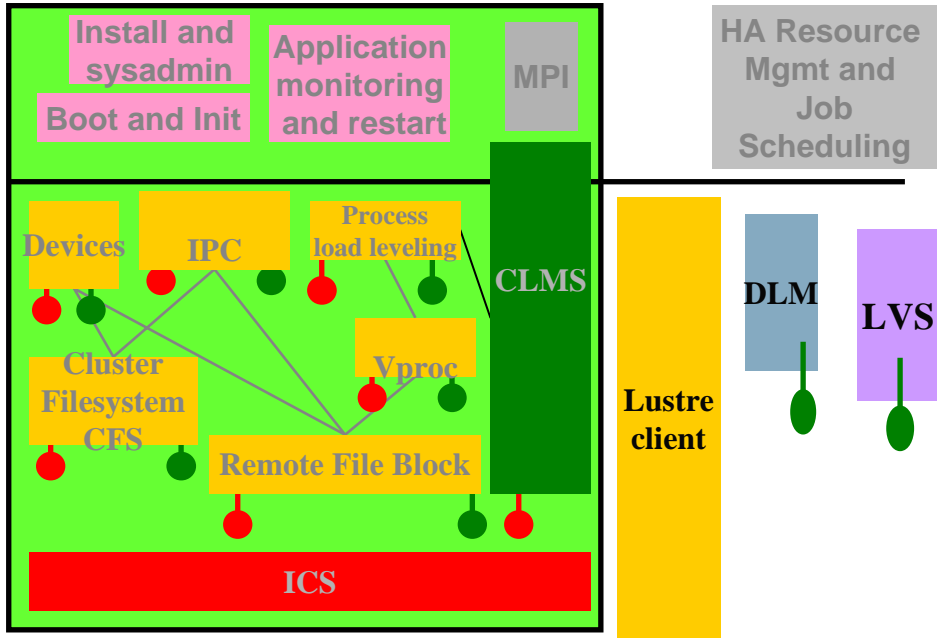
- Booting and Node Join
- Monitoring
- Filesystem
- Load balancing (process and connection)
- DLM
- IPC

Solution for Very Large Scale

Have 2 types of node – Service nodes (<200) and compute nodes (>1000)

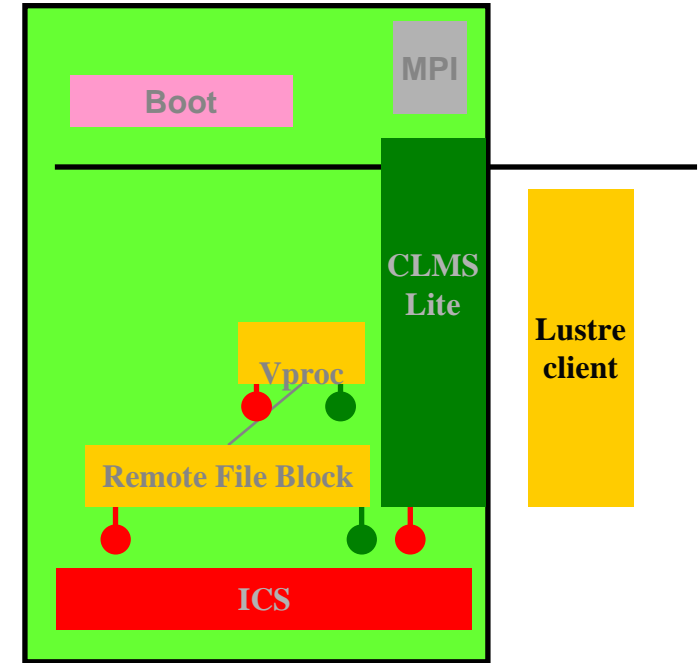
- service nodes have all the existing OpenSSI capability
- compute nodes are SSI but have some limitations

Approach for researching PetaScale SSI



Service Nodes

- single install;
- local boot (for HA);
- single IP (LVS)
- connection load balancing (LVS);
- single root with HA (Lustre);
- single file system namespace (Lustre);
- single IPC namespace;
- single process space and process load leveling;
- application HA
- strong/strict membership;



Compute Nodes

- single install;
- network or local boot;
- not part of single IP and no connection load balance
- single root with caching (Lustre);
- single file system namespace (Lustre);
- no single IPC namespace (optional);
- single process space but no process load leveling;
- no HA participation;
- scalable (relaxed) membership;
- inter-node communication channels on demand only

What about Scaling much higher

Potential Problem Areas:

- Booting and Node Join
 - Parallel multicast boot and CLMS lite for node join
- Monitoring
 - CLMS lite does monitoring less often and without load balancing info
- Filesystem
 - Use Lustre for data and special pseudo filesystems for proc and dev
- Load balancing (process and connection)
 - No load balancing for compute nodes
- DLM
 - No DLM on compute nodes
- IPC
 - If necessary, no clusterwide ipc on compute node?

OpenSSI Linux Clusters - Status

Version 1.2 released in December 04 –

Linux 2.4.22 based; stable branch

Binary, Source and CVS options

Functionally complete FC2, Debian, RHEL3

IA-32 and Itanium Platforms

Runs HPTC apps as well as Oracle RAC

Version 1.9.1 recently released

2.6.10 linux kernel development release

Proposed Kernel Hooks

clusterproc hooks, etc.

Available at OpenSSI.org

Backup

1. SSI Cluster Membership (CLMS)

CLMS kernel service on all nodes

CLMS Master on one node

- **(potential masters are specified)**

Cold SSI Cluster Boot selects master (can fail to another node)

- **other nodes join in automatically and early in kernel initialization**

Nodedown detection subsystem monitors connectivity

- **rapidly inform CLMS of failure (can get sub-second detection)**
- **excluded nodes immediately reboot (some integration with STOMITH still needed)**

There are APIs for membership and transitions

1. Cluster Membership APIs

cluster_name()

cluster_membership()

cluster_node_num()

cluster_transition() and cluster_detailedtransition()

- membership transition events

cluster_node_info()

cluster_node_setinfo()

cluster_node_avail()

Plus command versions for shell programming

Should put something in /proc

2. Inter-Node Communication (ICS)

- **Kernel to kernel transport subsystem**
- **runs over tcp/ip**
- **Structured to run over other messaging systems**
 - Native IB implementation ongoing
- **RPC, request/response, messaging**
- **server threads, queuing, channels, priority, throttling, connection mgmt, nodedown, ...**

2. Internode Communication Subsystem Features

- Architected as a kernel-to-kernel communication subsystem
- designed to start up connections at kernel boot time; before the main root is mounted;
 - could be used in more loosely coupled cluster environments
 - works with CLMS to form a tightly coupled (membershipwise) environment where all nodes agree on the membership list and have communication with all other nodes
- there is a set of communication channels between each node; flow control is per channel (not done)
- supports variable message size (at least 64K messages)
- queuing of outgoing messages
- dynamic service pool of kernel processes
- out-of-line data type for large chunks of data and transports that support pull or push DMA
- priority of messages to avoid deadlock; incoming message queuing
- nodedown interfaces and co-ordination with CLMS and subsystems
 - nodedown code to error out outgoing messages, flush incoming messages and kill/waitfor server processes processing messages from the node that went down
- architected with transport independent and dependent pieces (has run with tcp/ip and ServerNet)
- supports 3 communication paradigms:
 - one way messages; traditional RPCs; request/response or async RPC
- very simple generation language (ICSgen)
- works with XDR/RPCgen
- handles signal forwarding from client node to service node, to allow interruption or job control

3. Filesystem Strategy

- Support parallel physical filesystems (like GFS), layered CFS (which allows SSI cluster coherent access to non-parallel physical filesystems (JFS, XFS, reiserfs, ext3, cdfs, etc.) and parallel distributed (eg. Lustre)
- transparently ensure all nodes see the same mount tree (currently only for ext2, ext3 and NFS)

3. Cluster Filesystem (CFS)

- **Single root filesystem mounted on one node**
- **Other nodes join root node and “discover” root filesystem**
- **Other mounts done as in std Linux**
- **Standard physical filesystems (ext2, ext3, XFS, ..)**
- **CFS layered on top (all access thru CFS)**
 - **provides coherency, single site semantics, distribution and failure tolerance**
- **transparent filesystem failover**

3. Filesystem Failover for CFS - Overview

Dual or multiported Disk strategy

Simultaneous access to the disk not required

CFS layered/stacked on standard physical filesystem and optionally Volume mgmt

For each filesystem, only one node directly runs the physical filesystem code and accesses the disk until movement or failure

With hardware support, not limited to only dual porting

Can move active filesystems for load balancing

4. Process Management

- **Single pid space but allocate locally**
- **Transparent access to all processes on all nodes**
- **Processes can migrate during execution (next instruction is on a different node; consider it rescheduling on another node)**
- **Migration is via servicing `/proc/<pid>/goto` (done transparently by kernel) or `migrate syscall` (migrate yourself)**
- **Migration is by process (threads stay together)**
- **Also `rfork` and `rexec syscall` interfaces and `onnode` and `fastnode` commands**
- **process part of `/proc` is systemwide (so `ps` & debuggers “just work” systemwide)**

4. Process Relationships

- **Parent/child can be distributed**
- **Process Group can be distributed**
- **Session can be distributed**
- **Foreground pgrp can be distributed**
- **Debugger/ Debuggee can be distributed**
- **Signaler and process to be signaled can be distributed**
- **All are rebuilt as appropriate on arbitrary failure**

Vproc Features

Clusterwide unique pids (decentralized)

process and process group tracking under arbitrary failure and recovery

no polling

reliable signal delivery under arbitrary failure

process always executes system calls locally

- **no “do-do” at “home node”; never more than 1 task struct per process**
- **for HA and performance, processes can completely move**
 - **therefore can service node without application interruption**

process always only has 1 process id

transparent process migration

clusterwide /proc,

clusterwide job control

single init

Unmodified “ps” shows all processes on all nodes

transparent clusterwide debugging (ptrace or /proc)

integrated with load leveling (manual and automatic)

- **exec time and migration based automatic load leveling**
- **fastnode command and option on rexec, rfork, migrate**

architecture to allow competing remote process implementations

Vproc Implementation

Task structure split into 3 pieces:

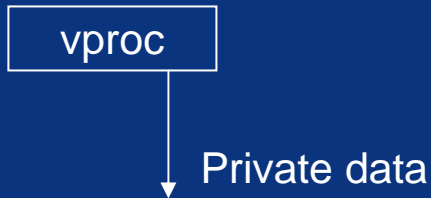
- vproc (tiny, just pid and pointer to private data)
- pvproc (primarily relationship lists; ...)
- task structure

all 3 on process execution node;

vproc/pvproc structs can exist on other nodes, primarily as a result of process relationships

Vproc Architecture - Data Structures and Code Flow

Data structures



Code Flow

Base OS code calls vproc interface routines for a give vproc

Define interface

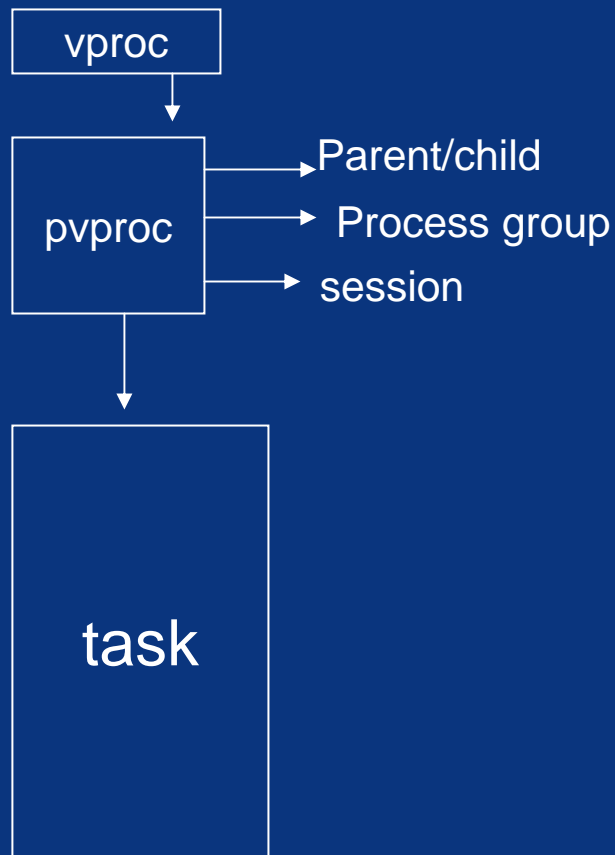
Replaceable vproc code handles relationships and sends messages as needed; calls pproc routines to manipulate task struct; may have it's own private data

Define interface

Base OS code manipulates task structure

Vproc Implementation - Data Structures and Code Flow

Data structures



Code Flow

Base OS code calls vproc interface routines for a give vproc

Define interface

Replaceable vproc code handles relationships and sends messages as needed; calls pproc routines to manipulate task struct

Define interface

Base OS code manipulates task structure

Vproc Implementation - Vproc Interfaces

High level vproc interfaces exist for any operation (mostly system calls) which may act on a process other than the caller or may impact a process relationship. Examples are sigproc, sigpgrp, exit, fork relationships, ...

To minimize “hooks” there are no vproc interfaces for operations which are done strictly to yourself (eg. Setting signal masks)

Low level interfaces (pproc routines) are called by vproc routines for any manipulation of the task structure

Vproc Implementation - Tracking

Origin node (creation node; node whose number is in the pid) is responsible for knowing if the process exists and where it is execution (so there is a vproc/pvproc struct on this node and a field in the pvproc indicates the execution node of the process); if a process wants to move, it must only tell it's origin node;

If the origin node goes away, part of the nodedown recovery will populate the "surrogate origin node", whose identity is well known to all nodes; never a window where anyone might think the process did not exist;

When the origin node reappears, it resumes the tracking (lots of bad things would happen if you didn't do this, like confusing others and duplicate pids)

If the surrogate origin node dies, nodedown recovery repopulates the takeover surrogate origin;

Vproc Implementation - Relationships

Relationships are handled through the pvproc struct and not task struct;

Relationship list (linked list of vproc/pvproc structs) is kept with the list leader (e.g.. Execution node of the parent or pgrp leader)

Relationship list sometimes has to be rebuilt due to failure of the leader (e.g.. Process groups do not go away when the leader dies)

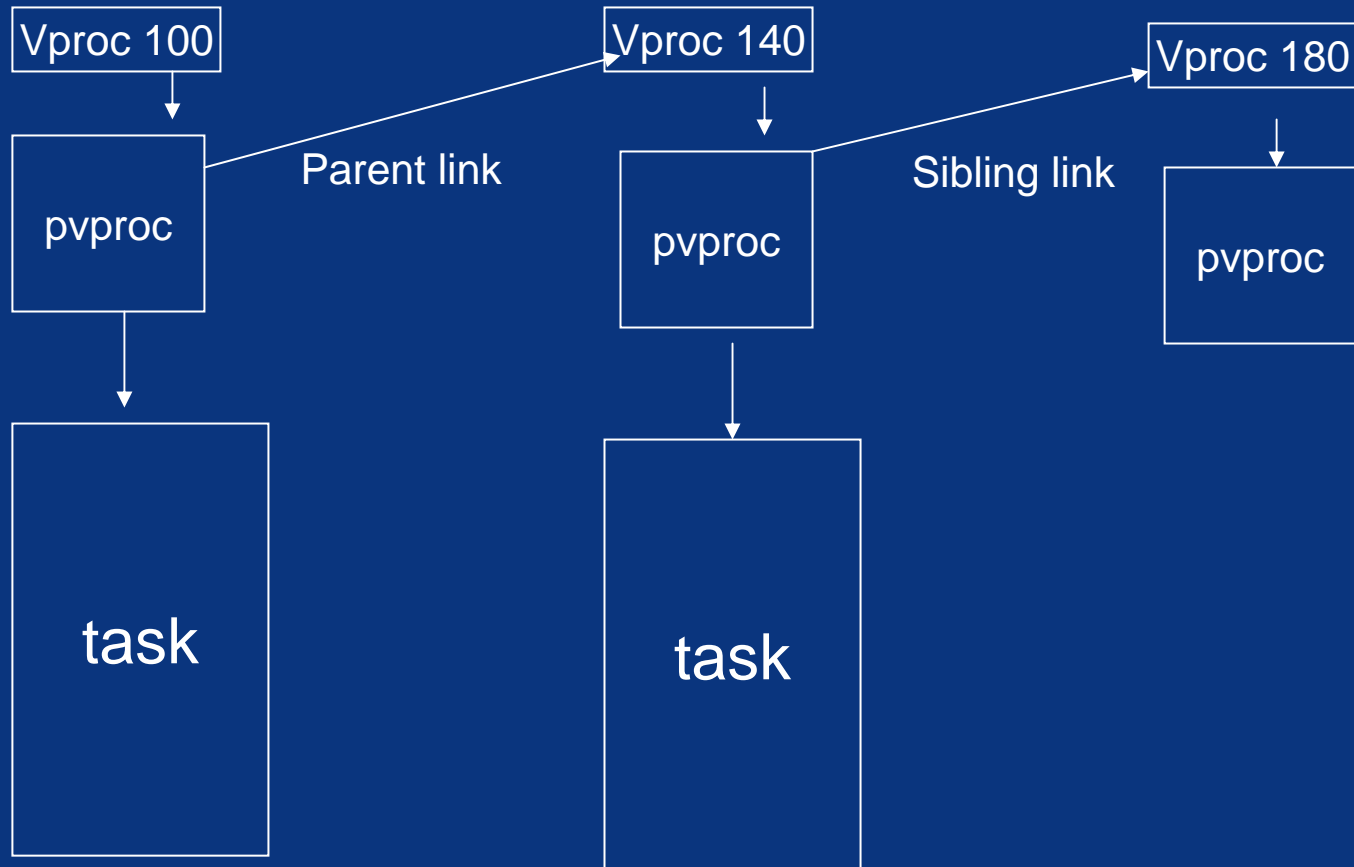
Complete failure handling is quite complicated - published paper on how we do it.

Vproc Implementation - parent/child relationship

Parent process (100)
at it's execution node

Child process 140 running
at parent's execution node

Child process 180
running remote



Vproc Implementation - APIs

rexec()- semantically identical to **exec** but with node number arg

- can also take “fastnode” argument

rfork()- semantically identical to **fork** but with node number arg

- can also take “fastnode” argument

migrate() - move me to node indicated; can do fastnode as well

- `/proc/<pid>/goto` causes process migration

where_pid() - way to ask on which node a process is executing

5. Process Load Leveling

- **There are two types of load leveling - connection load leveling and process load leveling**
- **Process load leveling can be done “manually” or via daemons (manual is onnode and fastnode; automatic is optional)**
- **Share load info with other nodes**
- **each local daemon can decide to move work to another node**
- **load balance at exec() time or after process running**
- **Selectively decide what applications to balance**

6. Interprocess Communication (IPC)

Semaphores, message queues and shared memory are created and managed on the node of the process that created them

Namespace managed by IPC Nameserver (rebuilt automatically on nameserver node failure)

pipes and fifos and ptys and sockets are created and managed on the node of the process that created them

all IPC objects have a systemwide namespace and accessibility from all nodes

Basic IPC model

Object nameserver
function
(track which objects
are on which nodes)

Object Server
(may know who the
client nodes are
(fifos, shm, pipes,
sockets,

Object client
knows where the
server is

7. Internet TCP/IP Networking - View Outside

VIP (Cluster Virtual IP)

- uses LVS project technology
- not associated with any given device
- advertise specific address as route to VIP (using unsolicited arp response)
- traffic comes in current director node and change nodes after a failure
- director node load levels the connections for registered services
- can have one VIP per subnet

7. Internet Networking

Scaling Pluses

- **Parallel stack (locks, memory, data structures, etc.)**
- **Can add devices and nodes**
- **Parallel servers (on independent nodes)**
- **Can distribute service**
 - **parallelization and load balancing**

9. Systemwide Device Naming and Access

- Each node creates a device space thru devfs and mounts it in `/cluster/nodenum#/dev`
- Naming done through a stacked CFS
- each node sees it's devices in `/dev`
- Access through remote device fileops (distribution and coherency)
- Multiported can route thru one node or direct from all
 - not all implemented
- Remote ioctls can use transparent remote copyin/out
- Device Drivers usually don't require change or recompile

13. Packaging and Installation

First Node:

- install Fedora, Debian or other distributions
- Run the OpenSSI “install”, which prompts for some information and sets up a single node cluster;
- Other Nodes:
 - can net/PXE boot up and then use shared root
 - basically a trivial install (addnode command)

14. Init, booting and Run Levels

- **Single init process that can failover if the node it is on fails**
- **nodes can netboot into the cluster or have a local disk boot image**
- **all nodes in the cluster run at the same run level**
- **if local boot image is old, automatic update and reboot to new image**

15. Single System Administration

Single set of User accounts (not NIS)

Single set of filesystems (no “Network mounts”)

Single set of devices

Single view of networking (with multiple devices)

Single set of Services (printing, dumps, networking*, etc.)

Single root filesystem (lots of admin files there)

Single install

Single boot and single copy of kernel

Single machine management tools

16. Application Availability

“Keepalive” and “Spawndaemon” part of base NonStop Clusters technology

Provides User-level application restart for registered processes

Restart on death of process or node

Can register processes (or groups) at system startup or anytime

Registered processes started with “spawndaemon”

Can unregister at any time

Used by the system to watch daemons

Could use other standard application availability technology (eg. Failsafe or ServiceGuard)

16. Application Availability

Simpler than other Application Availability solutions

- **one set of configuration files**
- **any process can run on any node**
- **Restart does not require hierarchy of resources (system does resource failover)**

OpenSSI Cluster Technology: Some Key Goals/Features

- **Full Clusterwide Single System Image**
- **Modular components which can integrate with other technology**
- **Boot time kernel membership service with APIs**
- **Boot time Communication Subsystem with IP**
 - (architected for other transports)
- **Single root; Cluster filesystem, devices, IPC, processes**
- **Parallel TCP/IP and Cluster Virtual IP**
- **Single Init; cluster run levels; single set of services**
- **Application monitoring and restart**
- **Single Management Console and management GUIs**
- **Hot-pluggable node additions (grow online as needed)**
- **Scalability, Availability and lowered cost of ownership**
- **Markets from simple failover to mainframe to supercomputer?**