# XCPU: a new, 9p-based, process management system for clusters and grids
# LA-UR-05-7562

Ron Minnich
Andrey Mirtchovski
Los Alamos National Lab, Los Alamos, New Mexico, USA

November, 2005

**Abstract**

Xcpu is a new process management system that is equally at home on clusters and grids. Xcpu provides a process execution service that presents itself to clients as a 9p-based service. It is visible to client nodes as a 9p server, and can be presented to users as a file system if that functionality is desired.

The xcpu service builds on our earlier work with the bproc system. Bproc distinguished itself from traditional remote execution services in several key ways, one of the most important being its use of a push rather than a pull model, in which the binaries are pushed to the nodes by the job starter, rather than pulled from a remote file system such as NFS. Bproc used a proprietary protocol; a process migration model; and a set of kernel modifications to achieve its goals. In contrast, xcpu uses a well-understood protocol, namely 9p; uses a non-migration model for moving the process to the remote node; and uses totally standard kernels on various operating systems such as Plan 9 and Linux to start, and MacOS and others in development.

In this paper, we describe our clustering model; how bproc implements it and how xcpu implements a similar, but not identical model. We describe in some detail the structure of the various xcpu components. Finally, we close with a discussion of xcpu performance, as measured on several clusters at LANL, including the 1024-node Pink cluster, and the 256-node Blue Steel infiniband cluster.

## Introduction: Science Appliance, the LANL clustering model

Science Appliance, a.k.a. Clustermatic, is a standard set of software components that, together, comprise the LANL clustering stack. Science Appliance includes

a BIOS, Linuxbios[]; a boot system, Beoboot[]; a system call which allows Linux to boot Linux, Two-Kernel Monte[]; software to allow single-point-control of all processes in the cluster, bproc[]; monitory software, Supermon[]; and a file system for resource sharing, v9fs[].

Global cluster process startup, monitoring, and control has, for 5 years now, been based on the bproc system, which was developed initially by Erik Hendriks while he was at NASA Goddard. While there are other key parts to Science Appliance, such as the use of the LinuxBIOS software, we will limit our discussion in this paper to bproc. The Science Appliance model has two key characteristics:

- A single control node for a cluster, from which all processes are started, viewed and managed. This node is called the master node. Users and system administrators perform all their cluster work on this single master node.
- A set of so-called slave nodes. Slave nodes have a very light kernel installation, with the root file system in a RAM disk. The RAM disk is regenerated each time the slave boots. A slave node has only a few processes, and only a few library and configuration files. No one ever logs into a slave node. All slave node control is performed from the master node.

Note that there is an inherent assymetry in this system. There are slave nodes, which know only of their own local resources; and a master node, which has a global view of the system. All processes in the cluster are visible on the master. All may be controlled from the master. Users log in to the master node only – there is no 'rsh' or 'ssh' which can log in to a slave node, since the slave node does not run those daemons.

From the master node, users have a single system image view of the entire process space of the cluster. Slave nodes, however, have a view of only their own process space. We call this type of system image 'Assymetric Single System Image', in light of the assymetry described above. The assymetry is essential to the scalability of the system. On Clustermatic systems, we have measured startup times for a 16 Mbyte MPI executable, on 1024 nodes, of 3 seconds. We know of no other cluster-based MPI startup that runs as fast. Symmetric SSI systems, such as OpenSSI, have never scaled to 1024 nodes, and even on smaller clusters, are far slower than bproc.

Symmetric Single System image software requires a great deal of inter-node communications to manage a global system image that appears the same on every node. By design, Symmetric SSI systems have a great deal of O(N^2) communications, as all the nodes must manage a view of all the other nodes. Assymetric SSI systems, in contrast, have O(N) communications. On Symmetric SSI systems, node failures can propagate from a single node to all the cluster. On Assymetric SSI systems, node failures are much better contained, as the failure is reduced to master-slave communications.

The Science Appliance model has proven to be very powerful. LANL has about 10,000 cluster nodes running this software, spanning approximately 20 clusters. The clusters vary in size, from 128 nodes to 2048 nodes. We estimate,

based on discussions with vendors, that there are several tens of thousands of cluster nodes running this software around the world.

Our experience with Science Appliance is that it provides greatly reduced costs, both in initial capital costs and ongoing maintenance. As compared to other similar clusters at our institution and elsewhere, we have found that we can typically reduce initial hardware costs by 50% using this software. We have also measured the ongoing cost in people to run our clusters, and we estimate that we have reduced those costs by a factor of 4 at minimum.

Our clusters are also far more reliable than others. We know of many cases where Science Appliance clusters have run unattended for over a year, continuously in use with a varying mix of jobs.

The current bproc component of Science Appliance, the component which provides the Assymetric Single System Image view of the cluster, has proven itself at LANL for supercomputing. At the same time, bproc is not without its limitations. The most serious, as we will describe below, is the need to patch the kernel to support the bproc model, but there are other issues as well. The bproc communications are managed via a non-standard protocol. Bproc communications are not secure; the assumption is that the network that supports bproc communications is somehow protected. Bproc does not seperate the process of moving the binary to a cluster node from the actual startup of that process; this lack of seperation has proven difficult for parallel debuggers and other, similar software.

Xcpu is intended to replace bproc, while preserving the bproc model as much as possible. Xcpu uses the v9fs, developed in part by LANL, and built into the kernel as of 2.6.14. Xcpu makes the same basic operations and model available to programs and users as bproc does, but in a more reliable manner, and without kernel patches of any kind. While xcpu has been used by LANL primarily for clusters, it has been also used in Grid environments, such as the 9grid; and it has even been ported to the Inferno distributed operating system.

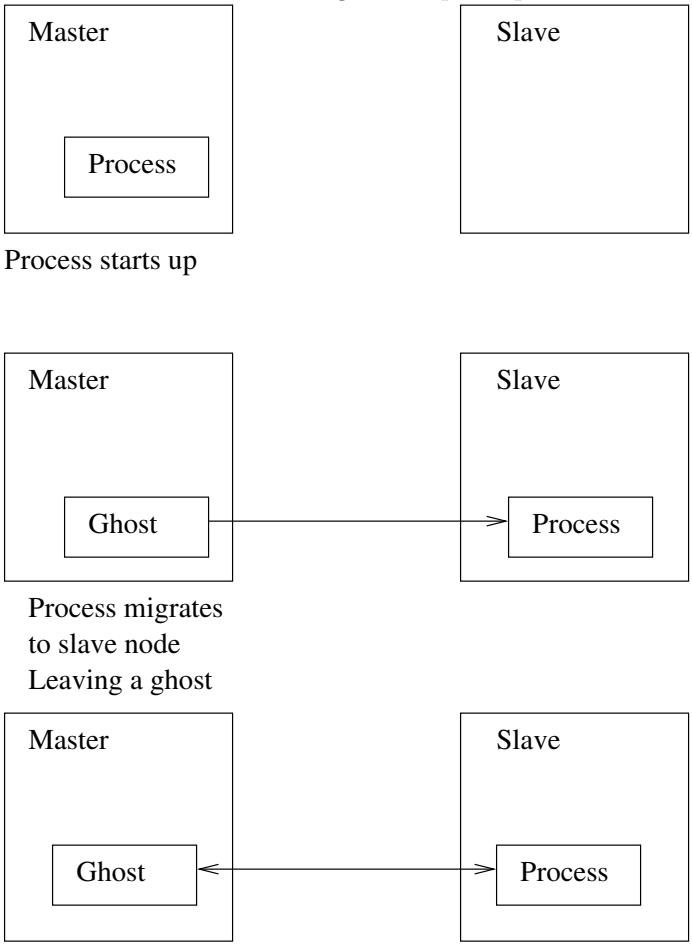
The rest of this paper is organized as follows. First, we provide an overview of how bproc works. Next, we explain the structure of xcpu, showing its similarites and differences from bproc. We describe the performance of xcpu on clusters of varying size, from 128 to 1024 nodes. We close with a description of some new ideas for how we plan to build xcpu clusters, including a new type of cluster model called 'Virtual Personal Supercomputing'.

# An overview of bproc

This description of bproc is of necessity brief. For more details, the reader is referred to [].

Bproc consists of a set of kernel modifications and user daemons that provide a single process space view of a cluster, from a single node called the 'master' node. All system calls which involve process creation, status, control, or termination, are modified such that the master node is capable of providing

Figure 1: bproc operation



a complete view of all the processes in a cluster. No user programs need to be modified in any way to participate in this single system image. Unmodified debuggers, for example, can transparently debug a process on a slave node.

The key to the bproc system is the extension of kernel process operations such that remote operations are redirected to a user daemon, which in turn relays the requests to a daemon on a slave node, which translates the request and issues the requests to the slave node kernel. In order to make the slave node process visible on the master node, 'ghost' processes populate the master node process table. These 'ghost' processes can be thought of as a link to the real slave node processes.

To create a slave node process, one must first create a master node process. This initial creation will also create, and initialize, the process table slot to be used for the ghost process. Once the process is created, it is checkpointed, transferred to the user-mode daemon on the master node, from there to the daemon on the slave node, and then recreated on the slave node itself, via an operation very similar to an exec.

We can thus see the basics of the bproc system: modified process system calls; modified kernel operations on processes; ghost processes on the master node which stand in for the real process on the slave node; and migration as the process startup mechanism on the slave node. The kernel modifications consist of a module which handles the checkpoint/restart, namely 'vmadump'; and the kernel patches to the process operations. The sum total of all this kernel code is about 25,000 lines. The daemons add less than 10,000 more lines. These minor changes to the kernel make every single command cluster-enabled; this seems a small price to pay for the addition of such a major capability.

Nevertheless, there are several issues that limit the long-term viability of bproc:

- Kernel patch. The kernel patch is very intrusive. It makes small, but significant modifications to such basic operations as fork. We feel it is unlikely that the bproc kernel patch would ever be accepted by the Linux core community, based on the initial feedback we received.

- /proc entry for ghost process is not complete. The /proc entry for a ghost process, while showing part of the remote process state, does not show all of the state. We have found that a lot of software, including TotalView, is confused by this partial information.

- bproc is too transparent. This problem was a surprise: it is possible to be too transparent. When users type 'ps', they transparently see all processes on the cluster; slave node and master node processes are indistinguishable, the kill command works identically for all processes, and so on. Not all users like this, especially sysadmins. We have found that users would rather have a ps that works for front-end processes only, and a ps *with a different name* that shows all cluster processes.

- bproc clusters have to be homogenous. It is not possible, for example, to build a mixed Opteron/G5 cluster, because the basic bproc startup

mechanism is process migration. We have even had problems with mixing different types of Pentiums in a bproc cluster. Use of process migration is proving to be a problem.

## Xcpu: maintaining, but improving, the bproc model

Xcpu is a new process management service, but the basic operations are similar to bproc. In xcpu systems, the binary resides on a *control* node. Computation is done on *compute* nodes, which themselves have no local binaries. To start a process on a compute node, a program moves the binary from the control node to the compute node – not, however, via migration. All the processes on a compute node may be viewed and managed from the control node, but for this purpose we use a standard u9fs server, found in both Unix and Plan9. On Unix, the u9fs server is modifed so that Plan 9-style process control commands (which are all textual) work over the network.

The xcpu programs include a process server, *xcpusrv*; a file system server, *u9fs*; and a set of client tools that run at user level, *xsh*, *xps*, and *xcp*. The xcpusrv and u9fs programs are 9p2000 servers; the client tools, since they access the service through the v9fs file system, are not special in any way.

Before we go into further detail about the server and client tools, we will provide a brief overview of the v9fs file system and the 9p2000 protocols.

### v9fs

V9fs is a new Virtual File System (VFS) for the Linux kernel. V9fs allows Linux kernels – and, hence, user processes – to access remote resources via the *9p2000* protocol[]. 9p2000 is a general-purpose resource sharing protocol used by the Plan 9 operating system[].

V9fs was written by Ron Minnich at the Sarnoff Corporation from 1996-1999, under contract to DARPA[]; udpated for use in 2.4 by Ron Minnich and Greg Watson at LANL in 2001[]; and, finally, brought into the 2.6 kernel tree by Eric Van Hensbergen of IBM in 2005[]. All Linux kernels past 2.6.14 include v9fs. While v9fs is built into the VFS layer, and is commonly used as a type of network file system, it is actually a resource sharing system. V9fs can be used to export services from one machine (server) to another machine (client) over almost any kind of network connection. The services are made available at the client machine as files: the nature of v9fs is such that the services, provided by servers, always look to the clients like a file system of some sort, although the actual nature of the service might be very different.

The xcpu server is a case in point. Xcpu provides a file-system-like interface to a remote execution service. Other possible services include a server that provides a file-system-like interface for the Web; a server that manages ssh connections, making them look like files; and a server that provides cluster monitoring and management service, much as Supermon does now.

**9p2000**

A full description of 9p2000 is outside the scope of this article. 9p2000 was originally designed for the Plan 9 operating system[]. It is the universal networking protocol "glue" used for all Plan 9 services, including file servers, window systems, DNS servers, security sytems, and so on. In a coarse sense, 9p2000 is very similar to the NFS[] protocol, with a few significant differences.

9p2000 uses a stream protocol, as opposed to the datagram protocol typically used by NFS. 9p2000 requires an ordered, sequenced, reliable transport which maintains the integrity of the data. Over this transport, 9p2000 has messages for attaching (mounting) a server; enumerating the files provided by the server (e.g., readdir); creating, opening, reading, writing, removing, and doing other operations on those files; and, in the end, detaching from a server. The interested reader is referred to [] for more information.

# Xcpu basics

We now cover the salient points of the lowest-level xcpu service. As mentioned above, Xcpu consists of user programs only; there are no kernel modifications needed. Xcpu client programs, if they use v9fs, require that kernel version 2.6.14 or later be used. Xcpu clusters consist of two types of nodes: control nodes and compute nodes. There is always at least one control node, possibly more; and at least one compute node, usually many more. Note that a control node can also be a compute node; one can develop and test xcpu on a single machine.

Xcpu servers need not run as root; the only privelege needed when running and using xcpu is the ability to mount a server in the client file system.

## Two classes of control nodes: system and user

There are two classes of control nodes: system and user. A system control node is capable of *managing* the xcpu service. A user node is capable of *using* the xcpu service to start processes, but can do no more. We have created the user control node class so that, in future, users might be able to attach Xcpu compute nodes directly to their desktop system, forming Virtual Private Supercomputers; while at the same time, the users can not take over the Xcpu server.

Virtual Private Supercomputing is a new capability provided by xcpu; it brings the nodes to the users, rather than forcing users to go to the nodes. To put it another way, the user will not log into the cluster control node to get access to the compute nodes, as they do now on existing clusters; rather, the cluster compute nodes will become a direct extension of the user's desktop workstation. We believe this capability will make many computation tasks easier for users.

Xcpu control nodes use a completely standard kernel, 2.6.14 or later, with v9fs support either compiled in or installable as a module. Control nodes gain access to xcpu services by attaching the compute node service into the name space of the control node – i.e., by mounting them. In every other way, Xcpu control nodes are standard.

## Two classes of compute nodes: compute and I/O

There are two classes of compute nodes, namely compute and I/O. The I/O node is no different in configuration as far as Xcpu is concerned, only in usage: I/O nodes are used to forward user file accesses from compute nodes to file servers, such as panasas. I/O nodes are used at LANL to aggregate file system activity, typicall from 16 nodes at a time, so as to reduce the impact of large clusters on our file servers.

Xcpu compute nodes use a completely standard kernel. There are no special modules needed to run a node as an xcpu compute node. Compute nodes run an xcpu server process, which exports the Xcpu remote execution service using the 9p2000 protocol. The only requirement for xcpu server processes is that some standard network transport – TCP/IP, usually – be available. The Xcpu server currently runs on Plan 9 and any system that supports the "plan9ports" software, such as Linux,Free-, Open-, and NetBSD, Mac OS X, Solaris, and many pther Unix and Windows systems. It is therefore possible to have an xcpu server consisting of, e.g., mixed Mac OS G5 and Linux Pentium, G5, and Opteron nodes.

## Security: xcpu can be used for distributed computing too

Xcpu is being used for distributed computing, on Plan 9, via the 9grid[]. Because Plan 9 uses a secure, encrypted, protocol for connecting hosts up, the problem of security is removed from the domain of Xcpu: accessing an Xcpu server, on Plan 9, looks like a standard Plan 9 mount operation. We plan to extend this secure distributed computing capability to Linux.

There is also a version of Xcpu being written in Limbo[], for the Interno[] operating system.

### Similarities and differences from bproc

There are some similarities between bproc and xcpu.

- No local binaries required. Both systems run executables transferred from a master, or control, node; initally, there are no binaries on the compute nodes, except for the ones needed to start it up.
- Local ram disk via initrd. Both systems use an initrd, installed into shmemfs, as the root file system.
- Control nodes sees all processes. The systems offer a view of all the processes running on the compute nodes to the control nodes – all the processes running on a cluster are observable on the control nodes.
- Control nodes control all processes. Control of all the processes is also possible from the control nodes – there is no need to log into a compute node to kill a process.

There are also many differences.

- Xcpu exposes operations. Where the basic operations of bproc were hidden, happening in inaccesible kernel code, in xcpu all the operations are exposed. Bproc grouped sets of operations together, and did them all at once for process startup; xcpu opens those operations up as individual file operations, making diagnosis much easier. In Xcpu, each step of starting up a process is exposed to the user program, allowing much finer control, which is very useful when ensembles of processes need to be started on a node, as for a parallel debugger.

- Xcpu uses standard protocol. Where bproc used a special protocol, xcpu uses the standard 9p protocol. Where each individual bproc operation involved a proprietary protocol operation, in xcpu those operations look like operations on files.

- No migration required; heterogeneity allowed. The bproc process spawn mechanism uses migration, requiring exact synchronization of architecture, kernel version, bproc version, and C library. Xcpu does not use a migration mechanism, allowing multiple architectures, kernel versions, and even xcpu versions. We have built clusters, that look much like bproc clusters, and yet consist of three different types of architectures.

- Multiple Xcpu servers allowed. Bproc required that one and only one bproc server run on a compute node; xcpu allows multiple versions if needed. This capability allows us to test newer xcpu servers while running old ones; and it also allows us to set up 'compartments' of different xcpu servers on a compute node for different user groups.

- Redundant control nodes allowed. Setting up redundant master nodes, in bproc, is worth a graduate degree; in xcpu, redundant master nodes are easy to set up.

- Can't use wait(2) on remote processes in Xcpu. Bproc allows wait system calls for processes on remote nodes; xcpu does not. We can hide this limitation in the programs that start up programs for users.

- Xcpu tolerates control node failure. Where failure of the bproc master meant a cluster reboot, xcpu clusters can tolerate a master reboot – compute node state is not lost if the control node reboots. To reattach the sessions, a backup control node can just remount the compute node service; or the control node, once rebooted, can reattach the service.

- Xcpu can support 'hot backup'. A hot backup configuration is possible, where two control nodes mount the same set of compute nodes, so that if one control node is lost, the other control node can take over.

## Xcpu server operation

In the following discussion we show Xcpu operation at the lowest possible level. In practice, users never see Xcpu servers at this level; instead, programs perform these operations.

## Xcpu operation

On compute nodes, an Xcpu server is started at boot time. Once the Xcpu server is up, it allows one or more connections from one or more control nodes, so that processes can be started.

For purposes of starting up a process, xcpu allows control nodes to establish *sessions*. A *session* is simply one instance of a conversation with the xcpu server, allowing a control node to start one process. Session startup is extremely low overhead, in fact, it is just the cost of opening and reading one file, as shown below. A session allows a remote user (or program) to start a process or process tree for one binary, once. In most cases, the process tree will be just one process, e.g. /bin/date; but it could also be a shell or other program which in turn forks more processes on a remote node. Thus, for a given session, xcpu will only start up one process; that process may, in turn, start up other processes. We have adopted this simple rule, one xcpu-started process per session, to simplify system management: it gives sysadmins more control of what users can do. If the scheduler or sysadmin wants to allow users arbitrary access to a machine, they can give the user access to the whole xcpu server; if, on the other hand, the scheduler wants to limit users somewhat, they can give the user access only to one session.

Note also that, as with bproc, access to both the session initation service and the sessions themselves can be controlled via *chmod,* since the session is initiated by access to a file, and each session is represented by a directory. One major difference here is that in bproc systems, the bpfs file system was optional, and bproc could run without it. In bproc, the bpfs file system was a way of reflecting the state of the bproc system to the user; in xcpu, the file system *is* the system.

To repeat, xcpu provides sessions via a session-initiation service. The session-initiation is presented, by the xcpu server running on a compute node, to the control node, as one file, namely, *clone*. Consider the case of an xcpu server mounted on a control node at, e.g., /mnt/xcpu/0/xcpu[1]. If we look at the file:

```
q:/home/rminnich # ls -l /mnt/xcpu/0/xcpu
total 0 -rw-rw-rw- 1 root     root 1969-12-
31 17:00 clone
q:/home/rminnich #
```

The clone file is the hook into establishing sessions with the xcpu server. To establish a new session, a process on the control node can open the clone file. This

[1] To see how this is achieved, see Appendix A.

creates a new session directory, with a new number. The number is available to the process by reading the clone file, viz.:

```
q:/home/rminnich # cat /mnt/xcpu/0/xcpu/clone
0q:/home/rminnich #
```

Reading clone returns the number of our session. Note that, as in most such services, there is no newline appended, hence the output – '0' – being butted up against the next command prompt. Again, this aspect of Xcpu operation would not be visible to users, only programmers.

The number actually represents the name of a directory, containing control and status files for that session. In this case, the directory name is 0, and an ls of /mnt/xcpu/0/xcpu/0 would reveal:

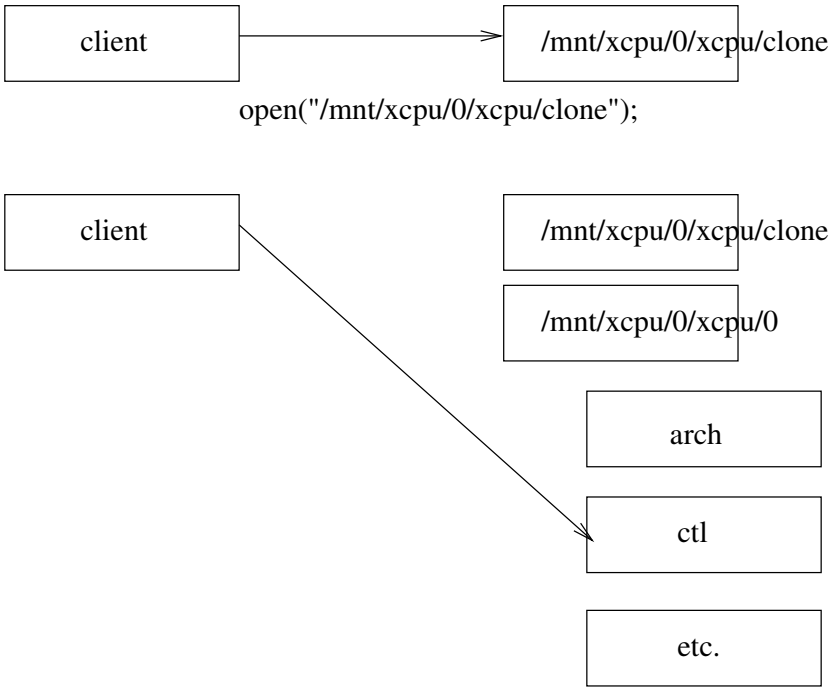
```
q:/home/rminnich # ls -l /mnt/xcpu/0/xcpu/0
arch  argv  ctl  exec  lastpid  sta-
tus  stderr  stdin  stdout  uname
q:/home/rminnich #
```

The files, in turn, are:

1. arch. The architecture of the machine, as a string, e.g. 'arm', 'i386', etc.
2. argv. The argv for the next command to be run.
3. stdin. Stdin of the process
4. stdout. Stdout of the process.
5. stderr. Stderr of the process.
6. ctl. Control for the session (session 0 in this case). The only command is exec, which will start the process. For a given session, this command can only be issued once.
7. exec. A write-once file containing the executable file image.
8. lastpid. Number of the last pid run (this will soon be deprecated)
9. status. Information about the state of this session.
10. uname. Uname of the machine, e.g. 'Linux', 'plan9', etc.

We show a picture of this operation below.

Figure 2: Running xcpu session

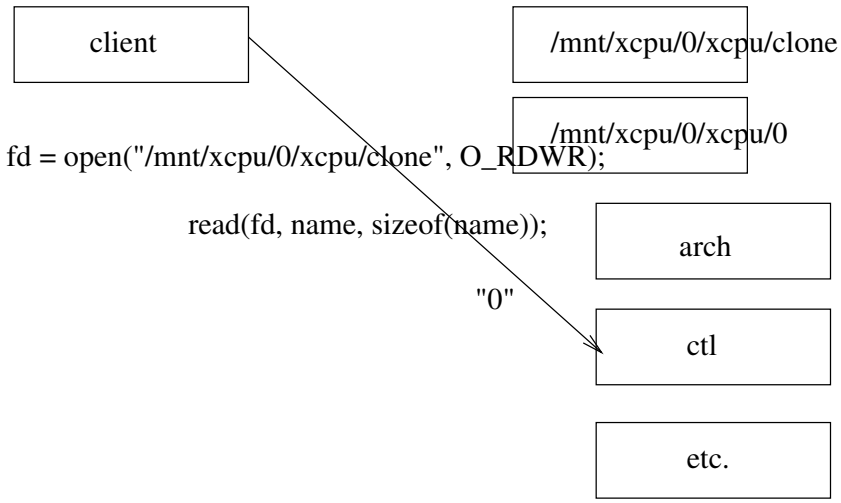



Compute node now has a session, #0. As more sessions are opened, the number increases. This sesison allows the user to copy one and only one program out and run it. Once the client closes /mnt/xcpu/0/xcpu/0/ctl, and all other files in the /mnt/xcpu/0/xcpu/0 dire the directory vanishes.

Client closes the file, and we see:


As shown in the Figure, opening and reading /mnt/xcpu/0/xcpu/clone results in the creation of a directory, populated with files. User programs never really open the clone file. Each time the Xcpu server sees an open for that file, it creates and populates a new session directory; establishes a new session struct; and returns, to the client, an open file descriptor for the *ctl* file in that new

Figure 3: Reading the file created by opening the clone file



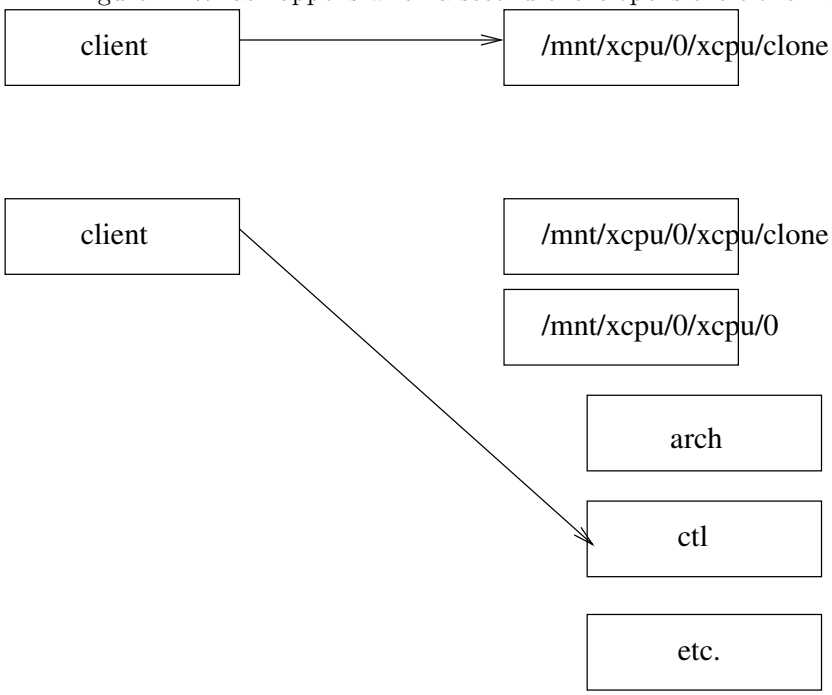

How does a client know which directory it has gotten?
By reading the file it opened, the client can read the
name of the directory

directory. This mechanisn is very commonly used in both Plan 9 file servers and drivers.

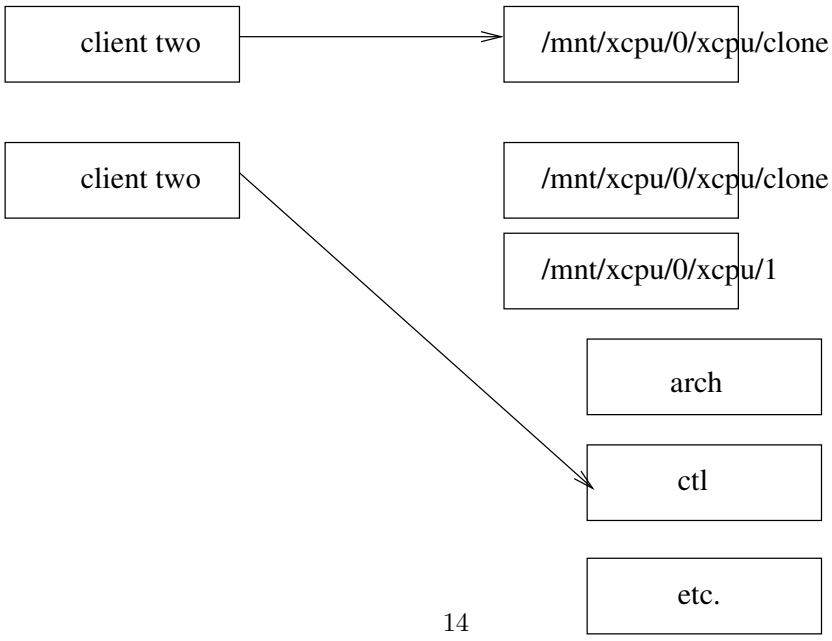
As each process in turn accesses the clone file, a new directory is created. One can think of the clone file as a factory for new sessions. The sessions which are created, and the directories which are visible to the control node, will persist as long as at least one of the files in the directory is opened. Once all the files in the directory are closed, the directory will vanish.

More than one client can open and read the clone file. Since, in every case, the clients don't actually ever have access to the clone file, there are no concurrency issues to worry about. There are also none of the race conditions that might occur if the clone file were a symbolic link, as is sometimes done in Linux systems (consider /proc/self). Figure shows the result of two clients opening the clone file. Please note, the clone file exists for each and every xcpu server, i.e. every compute node, accessible to the control node. The clone file, as such, resides on the compute node.

Figure 4: What happens when a second client opens the clone file




When second client opens /mnt/xcpu/0/xcpu/clone, the client sees a differe
If /mnt/xcpu/0/xcpu/0 and /mnt/xcpu/0/xcpu/1 are mode 0700, then clients
see each other's sessions



## Running a process

Once we have a session, i.e. we have a directory such as /mnt/xcpu/0/xcpu/0, then we can run a command.

First, however, we need the session around long enough to run a command. In the example, the cat will open and close the clone file, and the new session will disappear. The session lasts only as long as one file in the directory is open. How do we manage this problem?

On the user side, we use the method used by cluster schedulers such as LSF, and BJS. A proxy process will gain access to the nodes, then fork a shell or other process to actually use the nodes. The names of the nodes will be available as the environment variable XCPU_NODES. As long as the proxy process is alive, the nodes are available. We show this process in Figure.

For xcpu, sessions are created by reading the clone file. A tool, named *xcpu*, opens the file(s), sets an environment variable (XCPU_NODES), and forks a shell. At that point, xcpu will wait for all children to exit, while leaving the *ctl* file open. The xcpu process's presence will ensure that the subprocesses can open and close session files as needed, and remove them of the burden of having to make sure that at least one file stays open. The session will not disappear until the proxy process ends. This proxy process is 76 lines of code, and is actually a modified version of the Plan 9 'cat' command.

### Simple invocation via shell script

The rest of this discussion assumes that the proxy process has been run, and that a shell has been forked with the XCPU_NODES variable set to the nodes that have been allocated. For now, for this prototype stage, we are just returning the nodes as a list of session numbers. For the real system, XCPU_NODES should a list of pathnames.

We're going to show process startup as users would never do it, just to make it clear how it works. In practice, there would be a tool which bundles this together.

An Xcpu server, to run a process, needs an executable file and (optionally) arguments to run. As in bproc, the control node supplies the executable file. To get the file to the node, simply copy it:

```
for i in $NODES
do
cp /bin/date /mnt/xcpu/0/xcpu/$i/exec&
done
wait
```

Of course, there might be multiple architectures/OSes in the mix. If the executables are stored in a path with structure, e.g. /OS_name/architecture/path, then the script changes as follows:

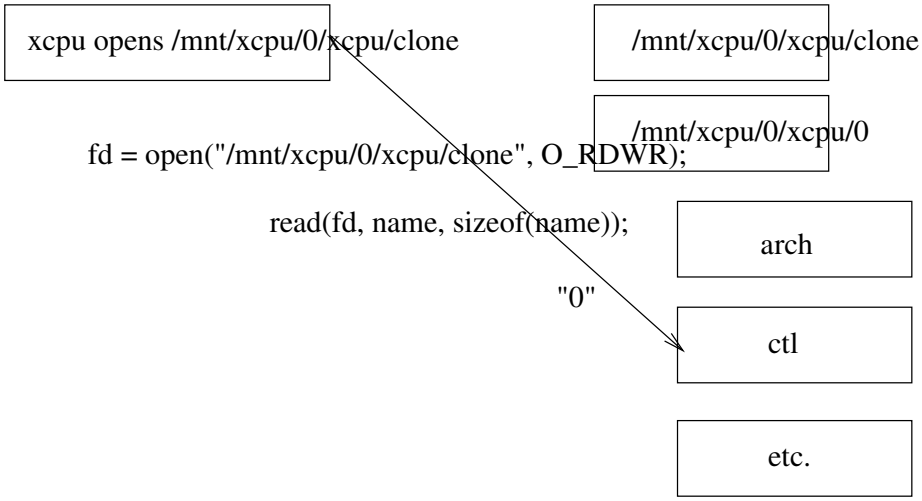
```
for i in $NODES
```

Figure 5: Operation of the tool used to set up sessions for xcpu users

User or program invokes xcpu command as:

xcpu /mnt/xcpu/0/xcpu

xcpu does the following:




xcpu puts the names of the directories into the environment

putenv("NODES", "0");

xcpu then forks and execs a shell:

```
if (fork() == 0)
   exec("/bin/sh");
```

and waits for the shell to exit

wait();

and then exits itself

exit();

Once xcpu exits, the last file is closed, and the directory vanishes as before

```
do
u=‘cat /mnt/xcpu/0/xcpu/$i/uname‘
a=‘cat /mnt/xcpu/0/xcpu/$i/arch‘
cp /$u/$a/bin/date /mnt/xcpu/0/xcpu/$i/exec&
done
wait
```

Note that some rather complex bproc kernel code has been replaced by a shell script. Note how trivial it is to support multiple architectures. There is a great deal of power in moving these concepts into the file system name space.

Next, argv:

```
for i in $NODES
do
echo date >/mnt/xcpu/0/xcpu/$i/argv&
done
wait
```

the argv is set up. Now to run it:

```
for i in $NODES
do
echo exec >  /mnt/xcpu/0/xcpu/$i/ctl
done
wait
```

and, get the output:

```
for i in $NODES
do
cat  /mnt/xcpu/0/xcpu/$i/console
done
wait
```

What if the process needs an EOF on stdin, for some reason? Since 9p, unlike NFS, has the concept of an open and close operation on the wire, we need merely open and close stdin, in order to propagate an EOF to the remote xcpu session:

```
for i in $NODES
do
echo -n ■■ >  /mnt/xcpu/0/xcpu/$i/stdin
done
wait
```

That's pretty much it; the entire bproc process, once all kernel code, is reduced to shell commands. All the control and status information is accessible via *cat*. Multiple architectures/OSes are easy to support. Errors are easy to track down. Console output can easily be redirected as needed. All the magic is gone. Making the operations of xcpu visible as file system operations has resulted in the elimination of complex, error-prone, kernel patches.

## Xcpu performance

The xcpu code is still in its first iteration through its life cycle and as such has not been optimized for speed. Still, even in this very early stage we established that the architecture does not lag significantly in speed from the more mature implementation of bproc. Our test environment consists of several dual Opteron machines connected via gigabit ethernet. One or more of the nodes act as xcpu servers (*compute nodes*) *with* each node starting a number of xcpu instances, which are mounted by the main execution node. A script runs a number of commands utilizing the xcpu architecture. It typically runs standalone binaries on instances of xcpu started on the compute nodes to measure how well the system scales, with all commands having with non-nil execution time and non-nil output. The following table shows results for a single compute node running different number of xcpu instances and a single client executing a 50K binary (/bin/`bin/date`) on the controller. The binary returning 30 bytes of output. In each case the result was averaged over a thousand runs. Note that the result for 10000 xcpu instances is significantly out-of line with respect to the others because we were reaching the limits of the kernels and physical memory on the machines, which caused thrashing.

| Number of xcpu instances | Execution time (seconds) |
|---|---|
| 10 | 0.1 |
| 100 | 0.9 |
| 1000 | 12.3 |
| 10000 | 180.10 |

We plan to present much more extensive performance results and discussion in the final edition of this paper, as measured on several clusters at LANL, including the 1024-node Pink cluster, and the 256-node Blue Steel infiniband cluster.

(note to reviewers: access to Pink is planned for Dec. 2005; we will include and discuss those results in the final paper, as well as any changes we had to make for performance).

## Conclusions

Xcpu is a new cluster process management system which extends the bproc model, and also exposes the basic primitives of process startup as operations in a file system. Xcpu uses the new v9fs file system, present in 2.6.14 and later kernels. Xcpu eliminates the "bproc patch", yet supports the most important capabilities of bproc.

In this paper, we discussed the design, structure, and implementation of Xcpu, and its performance on a 256- and 1024-node cluster.

Figure 6: /etc/xinetd.d/u9fs

```
# description: u9fs server
service 9pfs
{
disable = no
socket_type = stream
wait = no
user = root
group = root
protocol = tcp
port = 564
server = /usr/local/bin/u9fs
server_args = -D -a none -u root -m 65560
}
```

# Appendix A: HOWTO

(note to reviewers: the latest information on how to use xcpu will be included here in the final paper. )

## Linux client (i.e. control nodes)

Xcpu control nodes require v9fs, and v9fs requires 2.6.14 or later kernels. To get these kernels, one must pull it down from kernel.org.

## Linux/Unix server (i.e. compute nodes) and xcpu program for control nodes

The programs for the compute and control nodes run on both Plan 9 and Unix systems. They are written using the Plan 9 Port library, a.k.a. "plan9ports". To get plan9ports, for now, one can download http://www.swtch.com://plan9port/plan9port.tgz. Untar that file and just run INSTALL in the directory it creates. One usually puts this directory in /usr/local/bin.

Xcpu source is available at http://xcpu.sandia.gov.

A real Linux/Unix/MacOS cluster environment will use xcpusrv for starting processes, and will use the *u9fs* server for node monitoring. U9fs is part of the v9fs tools, but for convenience, we have added it to the xcpu directory. One must cd to u9fs and type 'make' to get it.

What follows is a simple example of a laptop that is the control node and the compute node. u9fs must be set up to run via xinetd,using an entry in /etc/xinetd.d for this purpose. Shown below is the file one must create, /etc/xinetd.d/u9fs

Make sure that the following lines are in /etc/services:

```
9pfs 564/tcp # plan 9 file service
```

```
9pfs 564/udp # plan 9 file service
```

Now, one can mount the "remote" node, i.e. the laptop. For convention, we're doing the following: compute nodes are accessed at /mnt/xcpu/<node #>/xcpu, for the xcpu server, and /mnt/xcpu/<node #>/fs, for the file system of the node. So, we do this:

```
mount -t 9P -o noextend 127.1 /mnt/xcpu/0/fs
```

Note that we can now see pretty much everything. We can see, for example, the /tmp file system for the node at /mnt/xcpu/0/fs/tmp. We don't need bpcp any more; to get a file to a node, we can just use cp. To see all the processes on the node, we just

```
cat /mnt/xcpu/0/fs/proc/*/status
```

Note what a change this is from past practice. The control node is mounting the compute node. The compute node is an open book; it is totally visible to, and controllable from, the control node.

Now we need to get the xcpu service up. First, we need to run the server:

```
./o.xcpusrv -9 -s 'tcp!*!20001' -D 255&
```

The -9 makes the server 'chatty'; it will dump all 9p2000 traffic. The -s names the service; in this case, the service name is tcp, from any address, on port 20001. The -D sets the debug level to 255; you're going to see lots of output!

Finally, we mount this service on the 'compute' node:

```
mount -t 9P -
o noextend,port=20001 127.1 /mnt/xcpu/0/xcpu/
```

You can now do all the examples shown above, and monitor the node.

Missing on the Unix side are the ability to have interactive console access; to kill the process; and several other bits, such as a a ps command that will show the process status on the compute nodes. These are all in the work, as are other tools.

## Plan 9 control and compute nodes

For Plan 9, all that is needed is the xcpu code. Get the code as before, untar it, and

mk 'CONF=p9' all

you now have a server for Plan 9. Plan 9 can serve as compute nodes (running the server) or as a control node (mounting servers).