Pushing TORQUE jobs in a chroot environment

D. H. van Dok

November 9, 2007

momchroot.tex,v 1.3 2007/11/09 12:47:31 dennisvd Exp

Contents

1 The idea

Batch systems may run multiple simultaneous jobs on a single node. Ideally, these jobs are as isolated from one another as possible and fairly share the available resources. The idea presented here provides *file system level isolation* between jobs, by running each job in a different chroot environment.

The `chroot(2)` system call changes the the entry point, or root (`/`), of the file system to another directory *for the calling process and all its children only*. The chroot'ed process will effectively be 'locked' in that subdirectory. In particular, a job that has been chroot'ed in a separate file system will not be able to fill up the `/tmp` directory of its parent.

It must be emphasized here that although chroots are often proposed as a security mechanism, that is not the aim of this idea. The security risks of pushing jobs into chroots are more or less identical to running the system without chroots.

This plan was executed on a Linux CentOS 4 [1] machine with the Torque 2.1.6 batch system client [2]. This version of Torque was chosen because it came with the gLite 3[1] middleware [3], but there are no gLite specific patches to it.

[1] gLite is developed within the European EGEE project [4]

This work involved no modifications or action on the Torque server, only on the worker nodes that accept and run jobs coming from the server.

Looking in closer detail to the plan, we make the observation that jobs expect, and can be expected to use, a sizeable portion of the free hard disk space for temporary storage. Each chroot environment should offer a portion of disk space that is isolated from the other chroots. A second observation is that this isolation is not just for simultaneous jobs, but also for jobs that run later in time. This means that hard disk space should not be oversubscribed: if the node is expected to harbour at most five simultaneous jobs, each chroot should get only a fifth of the total free disk space that is allotted to jobs. It also means that disk space needs to be recycled between jobs.

The chroot environments will have to look like normal file systems, so including the usual directories like /usr, /opt, /tmp, /etc.

2 Setup

The plan has two main portions of work:

1. setting up the base system provisioning for several chroot environments (the easy part),
2. modification and configuration of Torque to set up a chroot environment for incoming jobs, do a call to chroot, and clean up afterwards (the difficult part).

2.1 The base system

The node was installed from scratch with CentOS 4 server and the gLite 3.1 middleware for Worker Nodes. The disk was configured to use Logical Volume Management (LVM), which gives some flexibility in the arrangement of disk volumes on a running system. (Another reason to use LVM is a nifty feature called LVM snapshotting, but we'll come to that.)

The test node was a dual CPU machine, and the Torque server was configured to send it at most two simultaneous jobs.

The single 46 GB hard disk was configured at system install time to have the usual /boot and swap partitions and a single large volume group of 41 GB. On this volume group a single 15 GB logical volume was created for the system files, leaving the remaining space in the volume group empty to be divided among the jobs later on.

2.2 Torque installation

The work on the Torque side was twofold:

- patch torque-mom to perform the chroot at the right moment
- write a pair of prologue/epilogue scripts to set up and tear down the chroot enviroments.

The standard gLite WN installation includes torque-2.1.6-2.1.6-1cri_sl4_2st at the time of this writing, so this was the version chosen for patching. The patch is not yet tried on other versions of Torque.

2.2.1 Patching Torque

Torque already has provisioned for using chroot. The way this is implemented is as follows: if the incoming job has a job variable named `PBS_O_ROOTDIR` the value of this variable is taken to be the root to chroot to. If this variable is absent or empty, the chroot is not performed.

The patch to Torque now does the following: for each job arriving at pbs_mom, check if there is a file ⟨*PBS_HOME*⟩/mom_priv/jobs/⟨*jobid*⟩.root, and if so, set the job `PBS_O_ROOTDIR` variable to the contents of that file. There was already a hook in the code where 'site specific job setup' actions could be placed, so `site_mom_jst.c` was the only file that needed to be changed. The full source of this file is in Appendix A.

Source RPMS were obtained from Steve Traylen's RAL pages [5]. The spec file was changed only to update the release number and to include the patch file.

The invocation of rpmbuild is

```
rpmbuild -bb torque.spec --with homedir=/var/spool/pbs
```

The other changes involve a pair of prologue/epilogue scripts, that will be run before and after a job. It now becomes the responsibility of these scripts to:

- set up a chroot environment
- publish the path to chroot to in the ⟨*jobid*⟩.root file
- clean up after the job finishes

2.3 Preparing the chroot environment

The initial approach was to use LVM snapshots of the base file system as a quick and easy way to have a starting point very nearly identical to the base system. This worked, but the performance was very disappointing.

A second approach was then tried by setting up fresh file systems with a lot of *bind mounts*; this also worked and did not have the drawbacks of the snapshot approach.

It could be argued that it's a waste of time to describe the first attempt in any detail at all, as it was obviously a failure. But we learn as much – if not more – from our errors as from our successes, so the example will serve the purpose of an educational deterrent.

2.3.1 The snapshot approach

One particularly fanciful feature of LVM is creating snapshots of logical volumes. A snapshot is like an instant copy of the original volume, and it initially shares all its data with the original. It can be treated as any other volume. If it contains a file system, it can be mounted read/write and used normally. But data written to the snapshot will only be visible in the snapshot, and data written to the original after the snapshot was taken will not be visible in the snapshot. This is done by a copy-on-write (COW) mechanism, where writes to either volume will cause the creation of a copy of the original block. Subsequent reads of the affected area in the snapshot will come from the copied block.

The maximum amount of storage to use for the COW space of a snapshot has to be allocated upon creation of the snapshot; if this space is filled no more changes can be written to the snapshot and it will be automatically disabled. If a file system happens to live on the snapshot at that time, it will probably become corrupted.

This mechanism sounded wonderful for setting up a chroot environment. The original system was installed on a 15 GB file system, and the volume group could host two snapshots

Table 1: Torque prologue/epilogue script arguments

argument	**prologue**	**epilogue**
argv[1]	job id	job id
argv[2]	user name	user name
argv[3]	group name	group name
argv[4]	job name	job name
argv[5]	resource limits	session id
argv[6]	queue	resource limits
argv[7]	job account	resources used by job
argv[8]		queue
argv[9]		job account

with 13 GB of COW space each. Because there is more than 2 GB on the filesystem in immutable files (the installed system software) it is theoretically impossible to overrun the COW space limit.

The prologue script in ⟨*PBS_HOME*⟩/mom_priv/prologue is an executable script that is run by root prior to running a job. Its counterpart is the epilogue script, run after every job. The arguments to the scripts are given in Table 1. See also the Torque documentation [6].

Here is a line-by-line discussion of the prologue script.

Prologue The script heads off with the typical boilerplate.

```
1  #!/bin/bash
2  PATH=/bin:/usr/bin:/sbin:/usr/sbin
3  export PATH
4  volgroup=VolGroup00
5  logvol=LogVol00
6  PBS_HOME=/var/spool/pbs
```

The chroot is given a unique name after the ⟨*jobid*⟩. The invocation of `lvcreate` allocates 13 GB from the same volume group as the original.

```
7  jobid=$1
8  lvcreate -L 13G -s -n $jobid /dev/$volgroup/$logvol
9  if [ $? -ne 0 ]; then
10     echo "could not create snapshot, sorry."
11     exit 1
12 fi
```

The chroot mount point is also named after ⟨*jobid*⟩.

```
13 root=/chroot/$jobid
14 mkdir -p $root
15 mount /dev/$volgroup/$jobid $root
16 if [ $? -ne 0 ]; then
17     echo "could not mount /dev/$volgroup/$jobid on /chroot/$jobid"
18     exit 1
19 fi
```

The original file system is not the whole story. Mounted file systems don't show up in the snapshot. So we have to mount proc, dev, sys and other mounts.

```
mount --rbind /dev $root/dev || echo "couldn't mount dev"
mount none -t proc $root/proc || echo "couldn't mount proc"
mount none -t sysfs $root/sys || echo "couldn't mount sys"
```

The home directories are automounted, but the job needs only the home directory of the user running the job. By reading the home directory we force the automount to happen, then we bind mount it in the chroot environment.

```
ls /home/$user  > /dev/null
mount --rbind /home $root/home
```

The last thing to do if all is well is to write the root file that triggers the patched `pbs_mom` to do the chroot.

```
echo $root > $PBS_HOME/mom_priv/jobs/$jobid.root
exit 0
```

Epilogue The corresponding epilogue script tears the place down. The beginning is mostly the same as the prologue.

```
#!/bin/bash
PATH=/bin:/usr/bin:/sbin:/usr/sbin
export PATH
volgroup=VolGroup00
logvol=LogVol00
PBS_HOME=/var/spool/pbs
jobid=$1
user=$2
root=/chroot/$jobid
```

Remove the root file.

```
rm -f $PBS_HOME/mom_priv/jobs/$jobid.root
```

Unmount the home directory. In the epilogue we're forcing these actions as we don't have another chance to recover.

```
echo "unmounting $root/home/$user ..." >&2
umount -f $root/home/$user || echo "failed" >&2
```

The other submounts are unmounted.

```
echo "unmounting $root/dev ..."  >&2
umount -l $root/dev || echo "couldn't umount dev" >&2
echo "unmounting $root/proc ..."  >&2
umount  $root/proc || echo "couldn't umount proc" >&2
echo "unmounting $root/sys ..."  >&2
umount $root/sys || echo "couldn't umount sys" >&2
```

The chroot environment is unmounted and the mount point is removed.

```
echo "unmounting $root ..." >&2
umount -f $root || {
        echo "failed to unmount $root" >&2
        echo "The current mounts are:" >&2
        mount >&2
}
echo "removing $root" >&2
rmdir $root || echo "failed to remove $root" >&2
```

At last, the snapshot is removed.

```
27 echo "removing logical volume ..." >&2
28 lvremove -f /dev/$volgroup/$jobid || \
29     echo "failed lvremove /dev/$volgroup/$jobid" >&2
30 exit 0
```

The elegance of this approach is that creation and cleaning up of the snapshot are done in constant time. That means that the prologue/epilogue have very predictable running times.

Initial tests with the snapshot approach seemed to agree with this theory, but later tests that wrote 5 GB of data to disk brought the system in dire trouble. The process was embarrassingly slow, and the job took nearly six *hours* to finish, roughly 25 times too long. Not only that, but the machine went nearly incommunicado during the time of the run. Any process requiring disk access was put on hold for minutes. Luckily the Torque monitoring process still helpfully reported the machine load to the server.

Later experimenting showed that LVM snapshots were to blame [7]. Even under the best of circumstances I could not get a better (or rather, less worse) degradation of write performance than a factor of 10.

2.3.2 The fresh file system approach

This was a disappointment, but setting up the chroot environment could be arranged in other ways. Snapshots schmapshots, why not create a real volume with a brand new file system? Creating a new filesystem and bind mounting everything but /tmp from the base system will create an environment that has all its free space dedicated to the user's job.

New prologue Here are the improved versions of the prologue/epilogue scripts.

```
1  #!/bin/bash
2  PATH=/bin:/usr/bin:/sbin:/usr/sbin
3  export PATH
4  volgroup=VolGroup00
5  logvol=LogVol00
6  PBS_HOME=/var/spool/pbs
7  user=$2
8  # chroot is /chroot/$jobid
9  jobid=$1
```

After the usual preamble, we create a new logical volume. The number of extents allocated is carefully calculated to be half the available extents.

```
1  lvcreate -l 3358 -n $jobid $volgroup
2  if [ $? -ne 0 ]; then
3      echo "could not create logical volume, please come back some other time."
4      exit 1
5  fi
```

Create a file system on it. This takes some 10 seconds on the system used.

```
6  mkfs.ext3 /dev/$volgroup/$jobid
7  if [ $? -ne 0 ]; then
8          echo "failed to mkfs.ext3 /dev/$volgroup/$jobid" >&2
9          exit 1
10 fi
```

The mount point is created.

```
root=/chroot/$jobid
mkdir -p $root || echo "could not mkdir $root"
mount /dev/$volgroup/$jobid $root
if [ $? -ne 0 ]; then
    echo "could not mount /dev/$volgroup/$jobid on /chroot/$jobid"
    exit 1
fi
```

At this point we have a bare file system, so it needs to be populated with all the known goodness.

```
for i in bin etc opt tmp var var/lib var/tmp var/spool/pbs dev home \
         lib proc sbin sys usr
do
        mkdir -p $root/$i || echo "could not mkdir $root/$i" >&2
done
```

The permission flags must be set so users can write to /tmp and /var/tmp.

```
chmod 1777 $root/tmp $root/var/tmp
chmod 755 $root/var
chmod 755 $root/var/spool
```

The necessary directories are bind mounted from the base system. We need /var/spool/pbs because the user's job resides there.

```
for i in bin etc lib opt sbin usr var/lib var/spool/pbs ; do
        mount --rbind /$i $root/$i || \
                echo "could not bind mount /$i on $root/$i" >&2
done
```

The remainder is identical to the original.

```
mount --rbind /dev $root/dev || echo "couldn't mount dev"
mount none -t proc $root/proc || echo "couldn't mount proc"
mount none -t sysfs $root/sys || echo "couldn't mount sys"
ls /home/$user  > /dev/null
mount --rbind /home/$user $root/home/$user
echo $root > $PBS_HOME/mom_priv/jobs/$jobid.root
exit 0
```

New epilogue The epilogue counterpart now goes like this. The changed part is highlighted.

```
#!/bin/bash
PATH=/bin:/usr/bin:/sbin:/usr/sbin
export PATH
volgroup=VolGroup00
logvol=LogVol00
PBS_HOME=/var/spool/pbs
jobid=$1
user=$2
root=/chroot/$jobid
rm -f $PBS_HOME/mom_priv/jobs/$jobid.root
```

```
11 echo "unmounting $root/home/$user ..." >&2
12 umount -f $root/home/$user || echo "failed" >&2
13 echo "unmounting $root/dev ..."  >&2
14 umount -l $root/dev || echo "couldn't umount dev" >&2
15 echo "unmounting $root/proc ..."  >&2
16 umount  $root/proc || echo "couldn't umount proc" >&2
17 echo "unmounting $root/sys ..."  >&2
18 umount $root/sys || echo "couldn't umount sys" >&2

19  for i in bin etc lib opt sbin usr home var/lib var/spool/pbs ; do
20         umount -l $root/$i
21 done

22 echo "unmounting $root ..." >&2
23 umount -f $root || {
24         echo "failed to unmount $root" >&2
25         echo "The current mounts are:" >&2
26         mount >&2
27 }
28 echo "removing $root" >&2
29 rmdir $root || echo "failed to remove $root" >&2
30 echo "removing logical volume ..." >&2
31 lvremove -f /dev/$volgroup/$jobid || echo "failed lvremove /dev/$volgroup/$jobid" >&2
32 exit 0
```

Initial testing with this approach showed that it worked. Most of the setup time was spent in creating the file system, which, considering that the file system size is constant, is a constant amount of time.

The load generating tests went fine, and setup/teardown of the chroot environments seemed to work consistently. This does need more intense testing to stir up hiding bugs.

2.4 Automount in chroot environments

While bind mounting goes a long way in providing a reasonably populated chroot environment, and pseudo file systems that live in the kernel such as proc, devpts, and sysfs can be mounted multiple times, there is one remaining difficulty: automounted (NFS) directories.

Suppose /data is automounted, so a request for /data/esia will trigger the lookup and mounting of an directory exported from an NFS server. The chroot environment could have a /chroot/data directory with a bind mount to /data. But what happens upon visiting /chroot/data/esia? This, through the bind mount, would trigger the automount daemon to create and mount /data/esia, but although the *created* directory `esia` shows up as /chroot/data/esia, the *NFS mount* does not appear there.

So bind mounts are no good in combination with automount. What are our options?

1. mount every thinkable NFS share in advance
2. give each chroots its own automounts

The first options seems downright wrong. The second option sounds like an enormous configuration hassle, but there is the possibility to let automount look at its own system and do a *bind* mount.

Consider the following case: /data is automounted from some NFS server far, far away. We create a chroot environment as per usual under /chroot/job17:12/. The file /etc/auto.chrootdata contains the following line:

```
* :/data/&
```

This is automount arcana meaning that *anything* (*) that is requested will be looked up under /data *on the same machine* under the same name (&).

An automount daemon is started with the following command:

```
automount /chroot/job17:12/data file /etc/auto.chrootdata
```

And now the two automounts cooperate. If the path /chroot/job17:12/data/esia is requested, this will trigger the new automount. It will try to look up /data/esia, which doesn't exist, but this triggers the original automount to immediately find and mount /data/esia from the remote NFS server. The new automount then bind mounts this to /chroot/job17:12/data/esia.

This scheme has already been tried and tested, because in an earlier attempt this was used for the home directories. This was later changed for the reason mentioned above lines 23–24 in the original prologue.

The following snippets of code show how to handle automount in the prologue/epilogue scripts.

Prologue snippet This is remarkably short. The automount process daemonizes itself so the program returns immediately.

```
1 automount $root/home file /etc/auto.chroothome
```

Epilogue snippet In the epilogue we have a little more work to do. The fact that automount pid shows up in the output of the mount command is used here.

```
1  pidofautomount=`mount | sed -n '/chroot\/'$jobid'\/data/ s/.*pid\([0-9]*\).*/\1/ p'`
2  if [ -z $pidofautomount ] ; then
3      echo "warning: could not determine pid of automount" >&2
4  else
5      echo "sending automount($pidofautomount) a polite request to finish..." >&2
6      kill -USR2 $pidofautomount || echo "failed to kill $pidofautomount" >&2
7      # give automount some time to finish
8      usleep 100000
9      if ps -p $pidofautomount > /dev/null ; then
10         echo "killing automount the hard way... " >&2
11         kill -KILL $pidofautomount || echo "failed to kill -9 $pidofautomount" >&2
12     fi
13 fi
```

3 Closing thoughts

It is shown above that pushing batch jobs in a chroot environment is feasible. Two slightly different techniques were tried: one using LVM snapshots, the other using a fresh file system. Snapshots suffered from severe performance loss. The other method has so far proven solid.

There is another reason why using snapshots is less preferable. Besides providing job isolation, chroot offers an opportunity to close off parts of the file system that jobs have no business

accessing at all, such as /var/log. The fresh-file-system-with-bind-mounts method can be carefully tailored to precisely the right amount of exposure of the base system, while the snapshot method bluntly copies the entire system.

The method here presented is not without drawbacks:

- it requires a patched version of Torque,
- it requires custom setup on worker nodes,
- failure to clean up after jobs – for whatever reason – will decrease the capacity of a node to set up other chroot environments,
- post-mortem analysis on misbehaving jobs is nearly impossible after cleanup.

As far as job isolation goes, there are many things that are still exposed:

- jobs live in the same process space (information leak),
- they compete for CPU cycles, and
- they compete for disk and network I/O.

There are alternative approaches to improve job isolation and resource sharing. The use of virtualization, with hardware support becoming ubiquitous, is a logical choice. There are at least two different flavours worth mentioning:

- full machine virtualization, which allows running of multiple full-blown guest operating systems;
- the Linux-VServer project [8] uses kernel-level isolation. The virtual units don't run their own kernel, but they get their own process table, disk space, and network.

There are many full machine virtualization implementations, such as Xen [9] or KVM [10] to name a few. They allow you to run multiple different virtual machines with different operating systems. Linux-VServer is more of a chroot on steroids.

References

[1] CentOS, the community enterprise operating system, www.centos.org.

[2] The Torque resource management system, http://www.clusterresources.com/pages/products/torque-resource-manager.php.

[3] gLite, lightweight middleware for grid computing, http://glite.web.cern.ch/glite/.

[4] The Enabling Grids for E-science project, http://www.eu-egee.org/

[5] RPM builds and sources for Torque, http://hepunx.rl.ac.uk/~traylens/rpms/torque/.

[6] Torque v2.0 Administrator's Manual, http://www.clusterresources.com/torquedocs21/.

[7] D. H. van Dok, LVM2 snapshot performance problems, http://www.nikhef.nl/~dennisvd/lvmcrap.html.

[8] Linux-VServer, http://linux-vserver.org/.

[9] the Xen® hypervisor, http://xen.org/.

[10] Kernel Based Virtual Machine, http://kvm.qumranet.com/kvmwiki.

A Torque patch

This is ⟨*torque source tree*⟩`/src/lib/Libsite/site_mom_jst.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pbs_config.h>   /* the master config generated by configure */

#ifndef UNSUPPORTED_MACH
#define PBS_MOM
#endif

#include <sys/types.h>
#include <pwd.h>
#include "portability.h"
#include "list_link.h"
#include "server_limits.h"
#include "attribute.h"
#include "job.h"
#ifndef UNSUPPORTED_MACH
#include "mom_mach.h"
#include "mom_func.h"
#endif

extern char *mom_home;

/* external function in request.c */
extern char *get_job_envvar(job *,char *);

int escape_rootpath(char *out, const char *in, size_t maxlen);

/*
 * site_job_setup() -  to allow a site to perform site specific actions
 *        once the session has been created and before the job run.
 *        Return non-zero to abort the job.
 */
int site_job_setup(pjob)
        job *pjob;
{
  // find if a chroot was created for this job
  // in the prolog; if so, set the job env var PBS_O_ROOTDIR
  // to point to it.

  // test if the file $PBS_HOME/mom_priv/jobs/$jobid.root
  // exists and use its contents as the root.

  attribute rootdir;
  FILE *rootfile; // the file that contains the chroot path
  char rootfilename[PATH_MAX]; // the name of said file
  char errbuf[PATH_MAX]; // for printing errors
  char chrootpath[PATH_MAX]; // the path to set PBS_O_ROOTDIR to
  char erootpath[PATH_MAX]; // escaped version of the above

```

```
51    if ( get_job_envvar(pjob, "PBS_O_ROOTDIR") != NULL ) {
52      // ROOTDIR already set; leave it alone.
53      return 0;
54    }
55    // ROOTDIR not set; set it now.
56    strcpy(rootfilename, mom_home);
57    strcat(rootfilename, "/jobs/");
58    strncat(rootfilename, pjob->ji_qs.ji_jobid,
59            PATH_MAX - strlen(rootfilename) - 6);
60    strcat(rootfilename, ".root");
61
62    rootfile = fopen(rootfilename, "r");
63    if (NULL == rootfile) {
64      if (errno == ENOENT) {
65        // perfectly harmless; file is not there
66        return 0;
67      }
68      snprintf(errbuf, PATH_MAX, "can't open(%s)", rootfilename);
69      return 1;
70    }
71
72    // we have a root file
73    if (fgets(chrootpath, PATH_MAX -1, rootfile) == NULL) {
74      perror("fgets");
75      return 1;
76    }
77    if (escape_rootpath(erootpath, chrootpath, PATH_MAX) == -1) {
78      fprintf(stderr, "erootpath too long\n");
79      return 1;
80    }
81    if (snprintf(chrootpath, PATH_MAX, "PBS_O_ROOTDIR=%s", erootpath) >
82        PATH_MAX) {
83      fprintf(stderr, "rootpath too long\n");
84        return 1;
85    }
86
87    decode_arst(&rootdir, NULL, NULL, chrootpath);
88    set_arst(&pjob->ji_wattr[(int)JOB_ATR_variables],
89             &rootdir,
90             INCR);
91    return 0;
92  }
93
94
95  //  escape commas and backslashes with backslashes
96  int escape_rootpath(char *out, const char *in, size_t maxlen)
97  {
98    const char *p = in;
99    char *q = out;
100   size_t c = 0;
101
102   while (*p != '\0' && c < maxlen) {
103     switch (*p) {
104     case ',':
```

```
105       case '\\':
106         // need to have room for at least 2 characters!
107           c++;
108           if (c  >= maxlen) break; // no more room
109           // there's room
110           *q++ = '\\';
111       default:
112         *q++ = *p++ ;
113         c++;
114       }
115     }
116     *q = '\0';
117     if (c < maxlen) return 0;
118     else return -1;
119 
120   }
```