# Attack of the Giant ANTS!

# Advanced Namespace ToolS

## for Plan 9 from Bell Labs

*Ben Kidwell, "mycroftiv"*

*ABSTRACT*

The Advanced Namespace ToolS are a collection of Plan 9 software designed for the following goals:

1. Reliability, Resiliency, Recoverability - make grids of Plan 9 machines tolerant to hardware and software failures.

2. Flexibility and Configurability - make it easy to change what role a system is playing in a grid, to add in new systems, and retire old ones, without disrupting the user environment.

3. Multiplexing namespaces on one machine - the standard Plan 9 installer sets up a Plan 9 environment with a namespace organized to appear similar to a traditional unix userland, and all software on the machine operates within this namespace. This is convention, not necessity, and the Plan 9 kernel can run multiple user environments with radically different namespaces on the same machine. One notable application of this is running both 9front userland and standard Bell Labs userland on the same box at the same time, one available via `cpu` and one via the terminal's physical display and i/o.

4. Ad-hoc namespaces built from multiple machines - The converse of the previous goal is the ability to easily create an environment built from diverse and distinct services on different machines. This is part of standard Plan 9 concepts but the existing tools and scripts don't provide many commands for en masse namespace operations to apply numerous mounts and binds to a group of processes quickly.

5. Data sharing and persistence - Multiple machines should be able to work easily with the same data, even dynamic, flowing data. Being able to study and replicate past data flows is useful.

6. Adminstration - It should be easy for the user to administer all the machines used to create the workspace. Administration should be contained away from the main user environment, but constantly and easily accessible.

7. Automatic backup/replication and recovery. Redundant usable copies of data should be made automatically, and entering an earlier or alternate environment should be non-distruptive and fast.

**I. Intro**

The goal is to present to the user a computing and data environment abstracted as far away from the underlying hardware as possible, by leveraging the power of namespaces and network transparency, and to make that computing and data environment as indestructible as possible. No matter what happens to the hardware of an individual machine or machines, the user can continue working with as little trouble as possible. The goal of no disruption must acknowledge some limits. A running application on a machine that dies will probably result in the loss of some state and perhaps some unsaved data, but this should not be more than a minor annoyance in the overall computing environment.

In standard Plan 9, a grid of systems will wish to make use of common fileservers over the network. TCP booted cpu servers are an excellent mechanism for a unified grid of machines, but achieving this maximizes their reliance on the network resources and the annoyance of multimachine failures resulting from problems on one node of the grid. Consider the case where a user at a terminal is connected via `cpu` and also making use of another fileserver on the network for their personal information. If the cpu's connection to the file server providing the tcp root goes down, what are the consequences for the user?

If the user at a terminal is using that cpu server, and also importing their $home from a different fileserver, the user's connection to their $home and working data set is disrupted, even though the $home fileserver is still fine - it is just the boot fileserver for the cpu server with the binaries that had a disruption. The failure of one system (the cpu's root fileserver with the binaries) has caused the disruption of 2 other working systems and ended the workflow of the user until the cpu is rebooted. The cpu kernel is still running fine - why does it need to reboot because a user-space program lost a file descriptor?

The ANTS software addresses these issues by removing the dependency of the kernel on the root filesystem. Once the kernel is loaded into memory, it can stand back from userland, and instruct userspace programs to dial each other and create a namespace for the user, but the main flow of control of the master process on the system should not depend on any file descriptor acquired from an outside system. That way, anything can die and the master flow of control never freezes trying to read from a dead fd.

The foundation of these namespace tools is a rewrite of the post kernel-load `boot` and `init` programs. The userspace is still created the same way, with a conventional root, but this namespace is launched separately from the core flow of control. The kernel is independent of any root and there is an always available service namespace which exists beneath the conventional user-facing namespace. The service namespace is a comfortable working environment with all software needed to begin and administer the machine and the userspace it is hosting.

Creating multiple independent namespaces means the user needs to be able to move easily between them and modify namespace in an extensive fashion easily. Moving between namespaces and restructuring namespaces is enabled with scripted namespace operations such as `rerootwin` to reroot a window without losing the connection to keyboard, mouse, and window system. Another kernel modification makes `/proc/*/ns` writable to control the namespace of any owned process. This allows scripted namespace operations at a higher level of abstraction such as `cpns` to make the namespace of one process a copy of another.

The user has control of persistent interactive shells in any namespace on any machine using `hubfs` and can pipe data freely between shells on different nodes. Access to these persistent shells is integrated into `grio` with the addition of a new menu command. Behind the scenes, the user's data is replicated between archival stores with `ventiprog` and multiple active copies of the current root fs can be produced almost instantly with `cpsys` of the root filesystem of one `fossil` fileserver to another.

The combination of these Advanced Namespace ToolS creates a user environment where multiple machines behave as a tightly integrated unit, but experience only minimal disruption when other nodes need to leave the grid. The user can keep working with their current data from whatever nodes are available as other nodes plug in or leave the grid. Current backups are always available on demand and the user can re-root to a new root on the fly. Persistent shells and text applications on any node are available from every other node. A detailed investigation of the components of the toolkit follows, beginning with the boot/service namespace.

**II. The boot namespace and later namespaces**

Here is the namespace of the master process on a rootless box:

```
bind #c /dev
bind #ec /dev
bind -bc #e /env
bind -c #s /srv
bind -a #u /dev
bind / /
bind #d /fd
bind #p /proc
bind -a #¤ /dev
bind -a #S /dev
bind /net /net
bind -a #l /net
bind -a #I /net
bind /bin /bin
bind -a /boot /bin
cd /
```

Nothing can kill this shell except the failure of the kernel itself. The tools compiled into `/boot` allow it to fork off new environments rooted to disk or network resources without connecting to it itself. This master process can be thought of as a "namespace hypervisor" which stays behind the scenes creating different namespace environments. The namespace of this master process is create by a rewritten and combined `boot` and `init` which sets up the kernel-only environment. Once this is done, the script `plan9rc` is forked off to take control of setting up a user environment of services on top of the kernel, but leaving the main flow of control unattached to those resources.

`plan9rc` is a script which acts as a general purpose launcher script for self-contained Plan 9 environments. It takes over the work usually done by the kernel to attach a root fileserver and launch a `cpurc` or `termrc.` It also provides an optional interactive mode which has offers complete user control of all bootup parameters. Once an environment has been launched it saves the parameters used to its local ramdisk, allowing the same environment to be spawned again non-interactively.

`plan9rc` is written to be compatible with existing installations and `plan9.ini` values as much as possible. It is focused on pc bootup and handles tcp booting, `fossil, venti, kfs,` and `cfs` in the same manner as the conventional kernel. `plan9rc` creates the foundation environment to allow `termrc` or `cpurc` to be run as usual, and an existing standard Plan 9 install should perform as usual when launched in this fashion. In this way the rootless kernel acts like a foundation lift for a building that lifts the entire structure and installs a new floor at ground level.

How does the user access and use this new foundation level to give us control over the system? This is done by a small cpurc-equivalent script named `initskel` that may be run by `plan9rc.` The `initskel` creates a miniature user environment rooted on a small ramdisk, so it is also independent of any non-kernel resources. This environment has a much richer namespace than the core kernel control process but it is still independent of any external resources. Compiled into `/boot` are enough tools to allow this namespace to run a `cpu` listener (on a non-standard port). Here is what the ns looks like after `cpu` in on port 17020:

```
bind  /root /root
mount -ac '#s/ramboot' /root
bind  / /
bind -a /root /
mount -a '#s/ramboot' /
bind -c /root/mnt /mnt
bind  /boot /boot
mount -a '#s/bootpaq' /boot
[ standadr kernel binds omitted ]
bind  /n /n
mount -a '#s/slashn' /n
mount -a '#s/factotum' /mnt
bind  /bin /bin
bind -b /boot /bin
mount -b '#s/bootpaq' /bin
bind -a /root/bin /bin
bind -a /root/bin /boot
bind  /net /net
bind -a '#l' /net
bind -a '#I' /net
mount -a '#s/cs' /net
mount -a '#s/dns' /net
mount  '#s/usb' /n/usb
mount -a '#s/usb' /dev
mount -c '#s/hubfs' /n/hubfs
mount -c '#D/ssl/3/data' /mnt/term
bind -a /usr/bootes/bin/rc /bin
bind -a /usr/bootes/bin/386 /bin
bind -c /usr/bootes/tmp /tmp
bind -a /mnt/term/mnt/wsys /dev
bind  /mnt/term/dev/cons /dev/cons
bind  /mnt/term/dev/consctl /dev/consctl
bind -a /mnt/term/dev /dev
cd /usr/bootes
```

This namespace is a very important namespace in the structure of the grid. It exists on every single machine, created underneath whichever `cpurc` or `termrc` they run. This environment is a perfectly user-friendly namespace, unlike the pure kernel namespace with no ramdisk attached. In fact, depending on what is compiled into the `bootpaq` and the optional `tools.tgz` contained in `9fat` which may also be added to the ramdisk, this environment, while slightly spartan (no manpages, only 1 or 2 fonts in `/lib` ) is in fact sufficient for many tasks. Furthermore, since the user is `cpu` in as usual to a new flow of control, the user can freely acquire new resources from here without fear. If the `cpu` environment breaks, it hasnt harmed the flow of control it spawned from, the service and utility namespace will be the same on next `cpu` in.

To aid in working using the service namespace as a base, scripts are provided to provide forms of re-rooting. Some of the simplest are `addwrroot` and `importwrroot` which target external file or cpu servers and acquire their resources and bind them in locally while still keeping the ramboot root. The binds are to acquire the `bin` , `lib` , `sys` , and `usr` directories from the remote system. If the user wishes to fully attach to a new root while maintaining a `drawterm` or `cpu` connection, the script `rerootwin` provides this functionality. This is one of the most important tools for fast transformation of a user sub-environment. `rerootwin` works by saving the active devices with `srvfs` of `/mnt/term` and `/mnt/wsys` , then it uses a custom namespace file to root to a named `/srv` or network machine, and then re-acquire the original devices from the `srvfs` to allow the user to remain in full control and continue to run graphical applications in that window. Here is what the namespace looks like after the user executes `cpu` into a service namespace, begins `grio` and then opens a window and runs `rerootwin` targeting a different machine on the network:

```
[ standard kernel binds omitted ]
bind /net /net
bind -a '#l' /net
bind -a '#I' /net
bind /net.alt /net.alt
mount -a '#s/slashn' /net.alt
mount -c '#s/oldterm.1005' /net.alt/oldterm.1005
mount -c '#s/oldwsys.1005' /net.alt/oldwsys.1005
bind /net.alt/oldterm.1005/dev/cons /dev/cons
bind /net.alt/oldterm.1005/dev/consctl /dev/consctl
bind -a /net.alt/oldterm.1005/dev /dev
mount -b '#s/oldwsys.1005' /dev
bind /mnt /mnt
mount -a '#s/factotum' /mnt
bind /root /root
mount -ac '#s/gridfour' /root
bind / /
bind -a /root /
mount -a '#s/gridfour' /
bind -b /root/mnt /mnt
bind /boot /boot
mount -a '#s/bootpaq' /boot
bind /bin /bin
bind -b /boot /bin
mount -b '#s/bootpaq' /bin
bind -a /386/bin /bin
bind -a /rc/bin /bin
bind /n /n
mount -a '#s/slashn' /n
mount -a '#s/cs' /net
mount -a '#s/dns' /net
mount -c '#s/hubfs' /n/hubfs
bind /mnt/term /mnt/term
mount -bc '#s/oldterm.1005' /mnt/term
bind /mnt/wsys /mnt/wsys
mount -bc '#s/oldwsys.1005' /mnt/wsys
bind -c /usr/bootes/tmp /tmp
cd /usr/bootes
```

Using the `rerootwin` script in combination with the service namespace makes the cpu server a true cpu server, because the user is no longer using the cpu's root at all. The cpu is only providing execution resources at the junction of two totally independent systems. By `cpu` into the service namespace and then `rerootwin` to different file servers, the user environment is equivalent to one rooted conventionally to that environment, but without the dependency. If the re-rooted environment breaks, the user's active workspace on the cpu outside the re-rooted window is unharmed.

The use of multiple independent namespaces, the ability of the kernel to launch and manage services without depending on a root fs, and provision of needed programs in the `bootpaq` and `tools.tgz` give us the foundation to make a highly reliable grid. How are services built on the platform the kernel provides to create the desired properties? (Reliability, redundancy, ease of maintenance and administration.)

**III. Redundant roots on demand: fast system replication and efficient progressive backup**

Two high-level scripts provide management of the grid's data flow via the service namespaces: `ventiprog` and `cpsys.` `ventiprog` is run either via a cronjob, or whenver the user wishes to update their backups. It is an efficient progressive backup script based on `venti/wrarena` so running the script more frequently simply means less data sent, more often. `cpsys` uses `flfmt -v` to duplicate the state of fossils between systems. By using first `ventiprog` to replicate data between ventis, then `cpsys` to clone a fossil via the rootscore, and then setting the `$venti` environment variable used by the fossil to one of the backup ventis, the user is given a current working copy of their environment with a completely different chain of hardware dependencies.

The preferred mode of operation is to run two ventis and two fossils, one per venti. One fossil and venti are assigned the role of 'main/future'. Data is backed up frequently between the ventis, and whenever desired, the user resets the rootscore of the 'backup/past' fossil. From their terminal, the user can keep working with their data if one leg of the system needs to be reset for whatever reason. In general the user will work on the main/future fossil (probably via another cpu) but has the backup/past available for scratch and testing. Because this fossil's data flow dead ends unless it is needed as a backup, it can be used for destructive tests.

A core concept is focusing on `venti` and rootscores as the essence of the user environment, not the on-disk `fossil` buffers. `Fossil` is thought of as a convenient way of reading and writing `venti` blocks, not as a long-term reliable storage system. The `fossilize` script takes the most recent rootscore and appends it to a file stored in the 9fat. Once a fossil file exists (usually as a drive partition) the `flfmt -v` operation is almost instantaneous. The use of frequent `flfmt -v` keeps fossils small and bypasses many issues historically associated with `fossil` and `venti` coordination. A valid rootscore in combination with multiple ventis hosting those datablocks means that any reliability issues with `fossil` on-disk storage has little impact on the user. Any fossil that 'goes bad' is simply `flfmt -v.` Only the integrity of the `venti` blocks is important, and `venti` and it's administrative tools have been reliable in this author's experience.

The early boot environment runs an `rx` listener to allow the data replication and other administrative tools to be executed easily from other nodes or via `cron.` Testing revealed an issue which compromised reliability in the case of failure: `factotum` tries to acquire `/srv/cs` , and the connection server is running in a standard rooted environment, if the connection server goes down, `factotum` will time out waiting for the connection server to help it authdial. To avoid this, one can either host `cs` and `dns` also in the "rootless" environment, or use `factotum` with the new `-x` option, which prohibits it from mounting a `cs.` In this case, `factotum` simply uses the auth server provided as a parameter with the `-a` flag.

In this way isolation of function and access of the ram/paq namespace from the standard user environment is established. This allows the `plan9rc` script to function as a namespace launcher which can start multiple `cpurc` or `termrc` on the same machine, each with a different root.

**IIII. User namespace management: multiple roots and writable /proc/*/ns**

Because the main flow of control launches the root environment using `newns` but stays separate, it is possible to run the `plan9rc` script multiple times to run the `cpurc/termrc` from different root fileservers. One example would be doing the initial `plan9rc` script in the manner of a tcp booted cpu server, serving a cpu envionment rooted on a remote fs, and then rerunning `plan9rc` and launching a terminal environment from a local disk fs.

An example of this flow is included in the `multiboot.txt` and `multibootns.txt` files. After the `plan9rc` script runs and sets up a normal tcp boot cpu server environment, the user issues the commands:

```
mv /srv/boot /srv/tcpboot      # standard namespace files look for /boot so make it available
interactive=yes                # if this value was not set by plan9.ini
plan9rc                        # run the plan9rc script and this time create a terminal environment
```

On the second run of the `plan9rc` script, the user answers "clear" to almost all prompts because those services and actions have already been taken. The user provides the new root from the local disk fs and chooses terminal to start the termrc, and now the machine initiates a standard terminal for the user. However, the tcp boot cpu namespace is still available. The user can `cpu -h tcp!localhost!17060` to the ram/paq namespace, then `rerootwin tcpboot.` Now if the user starts `grio` and maximizes it, the user has a namespace exactly identical to `cpu` to a remote tcp boot cpu server attached to a remote file-server - except it is was created by `cpu` into another namespace hosted on the local terminal. One interesting fact to note is that due to the `mv` of the `/srv` , unless the user has changed the `/lib/namespace` files to non-default settings for the boot/root mounts, the `cpu` listener started by the `cpurc` now results in `cpu` into the terminal namespace, because that is what is located at `/srv/boot.`

To demonstrate that these principles work for even more strongly diverging namespaces, a test of using `plan9rc` to launch both 9front and Bell Labs user environments simultaneously was conducted. Both can coexist on the same machine as normal self sufficient environments without competing and the user can even create a mixed namespace that has elements of each.

This points to the next component of the toolkit for working in and controlling divergent namespaces - the writable `/proc/*/ns` kernel modification and the `addns subns` , and `cpns` scripts. With processes operating in many different namespaces, it may be useful or necessary to modify the mounts and binds of running services - but most services do not provide a method for doing so. From a shell the user can issue namespace commands, and some programs such as `acme` provide tools (Local) to change their namespace, but as a general rule standard plan 9 only allows the user to actively modify the namespace of shells, the "system-wide" namespace of services remains mostly constant after they are started.

The writable ns provides a simple and direct mechanism to allow modifications of the namespace of any process owned by the user, including processes on remote nodes via import of `/proc.` Simply writing the same text string as used by the namespace file or interactive shells to `/proc/*/ns` will perform the namespace modification on that program equivalent to it issuing that command itself. In this way the ns file becomes more tightly mapped to the process namespace. The action of writing namespace commands to the namespace file with `echo` commands is simple and natural and provides full generality. The exception is mounts requiring authentication, which are not performed. This restriction can be worked around by creating a `srvfs` of any authentication-required mounts so the non-authed `/srv` on the local machine may be acquired.

The generality of this mechanism allows it to be used as the foundation for another level of abstraction - scripts which perform namespace operations en masse on target processes. The `addns` , `subns` , and `cpns` scripts perform simple comparisons on the contents of process namespaces and make modifications accordingly. It should be noted that the scripts in their current state do not parse and understand the full 'graph/tree' structure of namespaces so their modifcations are somewhat naive. This is not a limit of the writable ns modification, more sophisticated tools should be able to do perfect rewrites of the namespace of the target process, but doing this requires understanding the dependencies of later binds on previous operations. The current scripts simply compare the ns files for matching and non-matching lines and use this to generate a list of actions. In practice, this mechanism is capable of performing even dramatic namespace modifications, and the user can always make additional changes or modify the actions of the script by using the `-t` flag to print actions without executing them. During testing, it was possible transform a running 9front userland into a Bell Labs userland by repeated `cpns -r` between processes that had been launched on different roots and by the respective `cpurc` of each distribution. The namespaces of `rio` and the running shells and applications were all remotely rewritten via the `/proc` interface to use a different root fs and to bring their namespace into conformance with Bell Labs namespace conventions.

It seems accurate to describe the modified boot system with ram/paq namespace and the plan9rc script as a "namespace hypervisor" because it supports multiple independent namespaces and allows travel between them. The writable ns mod enables fine grained control over the namespace of every process owned by a user on an entire grid of machines.

The final component used to bind the diverse namespaces together into a controllable and usable environment is the persistence and multiplexing layer provided by hubfs and integration into a modified `rio` named `grio.`

**V. Hubfs and grio: persistent rc shells from all nodes and namespaces and multiplexed grid i/o piping**

The ANTS toolkit is designed to create different namespaces for different purposes. The top layer is a modified `rio` named `grio` which integrates with `hubfs`. The modification is simple: the addition to the menu of a `Hub` command, which operates identically to `New` except the `rc` in the window is connected to a `hubfs`. It is intended that each node on a grid, and possibly different namespaces on each node, will connect to the `hubfs` and create a shell with `%local`. In this way, shells from every machine become available within one `hubfs`.

To make this environment available to the user by default, a few commands can be added to `cpurc` and the user profile. One machine within a grid will probably act as a primary "hubserver" and begin a hubfs for the user at its startup. Other machines will 'export' shells to that machine, using a command such as

```
cpu -h gridserver -c hub -b srvname remotesys
```

The user adds a statement to profile such as:

```
import -a hubserver /srv &
```

When grio is started, it looks for `/srv/riohubfs.username` to mount. This way, whichever node the user cpus to will have the same `hubfs` opened from the `Hub` menu option in `grio` , and because all systems are exporting shells to the hub, the user can `cpu` to any node and then have persistent workspaces on any machine. The state of the hubs remains regardless of where and how the user attaches or unattaches.

The `initskel` script also starts a `hubfs` by default in the early boot environment. This allows the user to easily access the ramroot namespace from the standard user environment. If the user desires, they could pre-mount the `/srv/hubfs` started at boot instead of the networked riohubfs to enable easy admin work in that namespace. It is even possible to create two layers of shared hubs - a shared administrative layer shared between machines running shells in the ram namespace, and another set of hubs in the standard namespace. In fact, these two layers can be freely mixed.

This is another way `hubfs` functions - to 'save' namespaces. If there is a namespace which is sometimes useful, but diverges from the main environment, it can be built with in a `hubfs` shell to be visited later at will. A single `hubfs` can provide a meeting point for any number of namespaces built on any number of machines and allow data to be pumped directly between processes file descriptors.

As a proof of concept, `hubfs` was used to create a 4 machine encrypt/decrypt pipeline. Machine A hosted a `hubfs` and created the extra hubfiles `encin encout decout`. Machine B then both mounted the fs and attached to it, and began running

```
auth/aescbc -e </n/aes/encin >>/n/aes/encout
```

Machine B mounted the hubfs, attached a local shell, and began running

```
auth/aescbc -d </n/aes/encout >>/n/aes/decout
```

Machine D mounted the hubfs and viewed the decrypted output of `decout`. Machine A also 'spied' on the encrypted channel by watching `/n/aes/encout` to see the encrypted version of the data.

As a proof of concept test of distributed grid computation and resiliency and interactivity under continuous load, the first draft of this paper was written simultaneously with running the aescbc encrypt/decryt test. At the time this section was concluded, the test had reached 7560 cats of `/lib/words` through the encryption filter, while simultaneously running `ventiprog` to mirror the venti data and maintaining additional persistent `hubfs` connections to all local and remote nodes, as well as preparing this document and using another set of hubs for persistent `emu ircfs` sessions, and performing multiple other tasks distributed across all grid nodes. ( `contrib/install` font packages, `vncv` connection to a linux box, etc.)

[ The test was briefly paused with no errors after 24+ hours of continuous operation and 8gb+ of cumulative data written through to take a few snapshots of the state of hubs. The test was stopped after 35 hours with no errors and 12314 loops and the data saved. ]

**VI. The sum of the parts: A case study in creating an always available data environment on a home grid**

I run my kernel and tools on all of my systems except those which run 9front, because I have not yet studied how to adapt my modifications for that distribution. Here is a description of how my grid is set up and how the tools described above fit together to give me the properties I want.

The main leg of services is a native venti, native fossil, and native tcp boot cpu each as a separate box. All run the rootless kernel and launch their services from the rootless environment, which I have `cpu/rx` access to on each, independent of any other boxes status or activity.

The primary backup leg of services is provided by a single linux box running a p9p `venti` duplicate and qemu fossil/cpu servers on demand. This `venti` is constantly progressively backed up from the main, and the qemu fossils are frequently `cpsys` refreshed to a current rootscore. If the main leg has trouble or needs to be rebooted for reasons like a kernel upgrade, I continue working via this p9p `venti` and attached qemus. They are also always available as part of my normal environment, not simply as emergency backup. I often keep the qemus tcp rooted to the main file server, but they can start a fossil rooted from the alternate venti at any moment to provide a copy of my root.

Additional remote nodes are hosted on 9cloud and are another "rootless labs" instance and 9front. There nodes are integrated primarily via `hubfs.` The labs node hosts a hub which is then mounted and attached to from within the main local hub, so it is a hub to hub linkup between the local and remote systems. This allows the local and remote grids to be reset independently without disrupting the state of the `hubfs` and shells on the other side of wan. A final wan component is another remote `venti` clone which also receives a steady flow of progressive backup and stores the current list of rootscores.

The main native cpu server is the primary `hubfs` server, with an `import -a` of its `/srv` in the standard user profile. This puts its `hubfs` as the default `hubfs` opened by `grio` , allowing each cpu node to provide access to a common set of hubs. Each machine exports a shell to the hubserver so I can sweep open a new Hub window and easily switch to a persistent shell on any node. A separate `hubfs` is run by the hostowner as part of the standard `initskel` script. `Hubfs` is also used to hold the `emu` client and server for `ircfs` or for `irc7` and general inter-machine datasharing when needed.

The user terminal is a native 9front machine, but the user environment is always built from grid services with the terminal functioning as just that. The main resources in the namespace are the two local CPU servers, which act as the central junctions by running applications, mounting fileservers, and hosting `hubfs.` The native cpu's `/srv` acts as the primary focal point for integrating and accessing grid services. All grid nodes except `venti` and auth provide `exportfs` so `/srv` and `/proc` of almost all machines can be accessed as needed. The writable `proc/*/ns` mod makes importing `/proc` an even more powerful and flexible operation for controlling remote resources. Being able to `cpns` to rewrite the namespace of remote processes allows for server processes to be rebound to new services or namespaces as they are available.

My data is replicated constantly with `ventiprog` , and I can instantly create new writable roots with `cpsys.` From any namespace on the grid, I can `rerootwin` to a new root and still maintain control with my active devices and window system. If any node has trouble, I can `cpu` into the service namespace with no dependencies on other services to repair or reset the node. Any textual application on any node can be persisted with `hubfs` to keep it active, out of the way but available for interaction if needed, and hubfs also can be used for distributed processing although I don't personally need to crunch many numbers.

All grid services are 'hot pluggable' and I can keep working with my current data if I need to reboot some machines to upgrade their kernels or just want to turn them off. All my services are constantly available and my namespace has no extra dependencies on services it isn't making use of. Cpus act as true 'platforms' to build namespaces because the user can work within the service environment and freely climb into any root with `rerootwin.`

All of these properties are based firmly on the simple core of Plan 9 - user definable per process namespaces, network transparency, and simple file access as the primary abstraction. The reconfigurations from the standard system are intended to focus and leverage these design aspects of the system. I am trying to extend Plan 9 in natural directions, using the existing code as much as possible, and just provide additional flexibility and control of the already existing capabilities.

**Appendix I: The pieces of the toolkit and how they serve the design goals:**

### bootup kernel mods, plan9rc, initskel, bootpaq, tools.tgz

These create a more flexible platform for namespace operations, and remove the dependency of the kernel on external services. They create a functional environment that acts as a minimal cpu server, and also can launch standard environments with a normal `cpurc` or `termrc`. The bootup process may be left almost unchanged in terms of user visible interaction, but the pre-existing installation now co-exists with the new boot "service/namespace hypervisor" layer.

### rerootwin, addwrroot, hubfs, savedevs/getdevs:

These allow the user to navigate namespaces easily, to attach to new roots, to "save" namespaces and shells for later use in `hubfs` , and to keep control of their current devices and window system while doing so. They are necessary to get the best use from the rootless environment, but they are not dependent on it. These namespaces control tools may be useful even without any changes to the kernel or boot process.

### writable proc/*/ns, cpns, addns, subns:

This kernel mod extends the power of `/proc` to modify the namespace of any processes owned by the user, on local or remote machines, simply by writing the same text string to the ns file of the proc that would be written in a shell. This mod is very general and powerful, but only `cpns` and its related scripts directly depend on it. I believe being able to do these namespace operations is a good part of Plan 9 design, but the other pieces of the toolkit are not written requiring this mod. The bootup sequence and `plan9rc` modifications are separable.

### ventiprog, cpsys, fossilize, /n/9fat/rootscor:

These scripts are written to help make use of the existing `fossil` and `venti` tools to improve reliability and let enable easy cloning of root filesystems and preservation of rootscores. If `venti` and `fossil` are being used, I believe these tools are at least a good model for how to manage them. There is no inherent dependency on the rest of the tools on `venti` or `fossil` , but the ability of `fossil` to instantly create a new root with `flfmt -v` is a powerful tool and many of my workflows are built upon it. The flow of `flfmt -v` , `fossilstart` , `rerootwin` into the new fossil can be done in a few seconds and provides a new interactive root environment that 'flows' directly from the old one without eliminating it.

### hubfs, grio:

`Hubfs` is listed again because it is also part of the upper user interface layer in addition to the lower network piping layer. The user can work easily in all their different namespaces because `grio+hubfs` makes access to persistent shells in diverse namespaces as easy as opening a `New rc`. The color-setting option of `grio` also lets the user 'organize' their namespaces by sub-rios with different colors.

These components are all separable, but I believe the whole is greater than the sum of the parts and so created the giant ANTS packages. It is possible to use `hubfs+grio` without changing bootup or namespaces, or possible to create a more reliable bootup and independent early namespace without using `hubfs` or `grio` , and the concepts of the `rerootwin` script may be generally useful independent of any tools at all. The goal is to provide a true toolkit approach to namespaces where the user can make the environment that serves them best.

**Appendix II: Implementation details:**

Boot mods: the goal is to create a working environment with only kernel resources, roughly speaking. This is pretty established territory, the main thing I have done differently than some other developers is to parameterize as much as possible and just not get the root fs! `Boot/init` are combined into a single program and most of their functionality is shifted to the `plan9rc` script, supported by a compiled in `bootpaq`. The `plan9rc` , `ramskel` , and `initskel` scripts work to make a minimal but working environment by gluing a skeleton `ramfs` to the compiled in `bootpaq`. Once this is done, a "root" fileserver can be acquired and its `termrc` or `cpurc` forked off into a `newns` where it becomes a working environment without taking over the flow of control in the kernel only environment.

Writable `proc/*/ns` : this was implemented by more or less 'cloning' the entire code path within the kernel that happens for mounts and binds and adding a new parameter for process id. All of the existing routines use "up" to figure out what namespace is being modified and what the chans are - by copying all of the routines and adding a new parameter, I allow the /proc file system to perform mount s and bind s on behal" of a process, acccording to the string written to that process's ns file. I made the mechanism use a copy of all the original routines with a new parameter because I didn't want my modifications to affect the existing code paths - especially because some sanity checks don't make sense if the context is not up and removing kernel sanity checks is scary. I have tested this mod extensively and I believe it is not inherently destablizing but it may pose unanalyzed security risks if abused by malicious users.

The cpns , addns , subns scripts perform their operations by comparing the lines of the textual ns files of the model and target processes, and issuing mount/unmount commands based on matching and non-matching lines. This mechanism is functional but better tools should be written, which fully understand how namespaces are structures as graphs with dependencies. Treating the ns files as text without understanding the real semantics of namespaces is a limitation of these scripts, not the writable ns mod that enables them.

Hubfs: hubfs is a 9p filesystem which implements a new abstraction which is similar to a pipe, but designed for multiple readers and writers. One convenient use of this abstraction is to implement functionality like GNU screen (http://gnu.org/software/screen/manual/screen.html) by connecting rc shells to hub files. The hubfs filesystem simply provides the pipe/hub files, the job of managing connections is done by the hubshell program, which knows how to start and attach rc to hubfiles, launch new connected rc shells on either the local or remote machine, and then move between the active shells.

Rerootwin: the rerootwin "device and wsys aware" re-rooting script and namespace file is based on a simple core technique: using srvfs to save the devices. The ability to control a given window and run graphical applications in it is simply a result of what is bound into the namespace. A standard newns command can't be used to enter a new root filesystem when working remotely, because the new namespace will not be connected to the device files of the previous namespace. The solution is to srvfs the devices first, make note of their identity in an environment variable, then enter the new namespace and re-acquire the original devices. This operation is basically simple and seems to have broad usefulness. I am actually surprised a similar script and namespace file does not already exist within Plan 9 because it does not depend on the other modifications in the toolkit.

The venti/fossil tools simply automate actions which are useful for backup and recreation, and the other namespace scripts mostly perform self-explanatory bind and mount operations. The modifications to rc and factotum are minimal and relatively insignificant. rc is modified only to path /boot and a different location for /rc/lib/rcmain , factotum simply adds a flag to prefer a code path which it had as a fallback previously, wrarena9 just adds output of the current clump as the data sending proceeds.

The hardware infrastructure is two native pentium IV for the main venti and fossil server and a pentium III for the main tcp cpu. The user terminal is a modern desktop with an intel i7 running the 9front distribution. An amd ph II debian box provides p9p and qemu hosting for the backup leg of services. Remote nodes are hosted on 9cloud with one Bell Labs and one 9front install. A linode running p9p provides a final fallback venti store.

**Appendix III: Origins of the tools**

I began working with multiple-machine Plan 9 system about 5 years ago, trying to experience the original design of separate services acquired from the network via a terminal. I have found this to be an environment with desirable properties, many of them as described in the original papers. I also encountered some obstacles as a home user trying to deploy a true distributed Plan 9 environment. In the original Plan 9 design, the infrastructure of file and cpu servers was intended to be installed in a professionally managed datacenter. The default assumptions, though somewhat adjusted over the years, remain best suited for a world where a given hardware and network configuration has a lifespan measured in years. In a home network, things may often change on a timescale of weeks, days, or even hours. The user is likely to turn off and turn on machines more often, shift both public and private nat ips, and in general operate the machines in a much less predictable way. Also, the hardware a home user is likely to be using for Plan 9 is a mixture of older machines, virtual machines, and desktops hosting related software like Plan9port. This is a very different reliability profile than professional datacenter hardware.

My experience as a home user building on top of older hardware mixed with virtual machines and making frequent changes to my network arrangement was that the user environment I had in Plan 9 was amazing, but somewhat fragile. The grid architecture created dependencies between the different machines. If there is a backing store machine (such as `venti` ) supporting a file server supporting a cpu server, the user environment breaks if there is any disruption of the machines and their network connections. At the time four years ago, Qemu virtualization also seemed less robust than now, and my VMs were prone to crashing if placed under significant load. Plan 9 was giving me a user environment that I loved, but I was also struggling with reliability issues - qemu VMs running `fossil` often corrupted their filesystem when crashing badly.

It seemed to me that a grid of multiple machines should create the property of being more reliable and resilient than any of the components, and I was experiencing more of a "one fall, they all fall" dynamic. The more tightly I tried to bind my machines together by importing services from each other, the more fragile everything became. I wanted the `cpurc` on the machines to acquire services from the other machines on the grid, to put those "underneath" the user namespace so that when I `cpu` in, a multiple machine namespace would be built and waiting. Doing this kind of service acquisition from the main flow of control in `cpurc` though created system wide dependencies on those resources, and my user environment would freeze up if a read from a network mount didn't return. I tried using `aan` and `recover` , but those tools are for dealing with network-level disruptions, not a machine that dies and has to reboot.

Another issue I experienced working with my grid was the lack of higher-level tools for dealing with namespaces, and a general focus on creating a namespace which roughly mirrored conventional unix. It felt to me that the mechanism of namespaces was so brilliant and so flexible and open-ended that there was much more that could be done with manipulating namespaces and network transparency to build interesting environments. What I wanted was a way to "weave" a complicated namespace that unified different machines, but was also resilient to their failure and would replicate and secure my data without extra work.

As I experimented with different partial solutions (just running 3 copies of everything, for instance) it became clear to me that there was a fundamental, and unnecessary, dependency that was at the root of my difficulties. This was the dependency of the Plan 9 kernel on a given root filesystem chosen at boot-time. When a running cpu server loses its root fileserver, it becomes dead - even though it experienced no failure or data corruption or any disruption in its normal function, it just lost a file descriptor. Deciding to restructure boot to remove this dependency was the core insight that the rest of the tools became organized around.

Mycroftiv, plan9grid@9gridchan.org

*Ben Kidwell, "mycroftiv"*

*Draft version 2, Feb 20 2013*