www.ueber.net /who/mjl/plan9/plan9-obsd.html

Running a Plan 9 network on OpenBSD(or unix in general)

27-34 minutes

intro

about plan 9

Plan 9 was designed to run as a network of machines: authentication servers, file servers, cpu servers and terminals. File servers just serve files over the network, cpu servers only execute programs (and have no local storage: all programs and user files are read from the file servers over the network), and the authentication server would keep it secure. Finally, terminals are the simple (again diskless) computers users sit at.

for new users

New users taking Plan 9 for a test ride generally do not want to install four machines before seeing what Plan 9 is all about. In practice, people run all three servers ("auth", "fs" and "cpu" for short) and the terminal on a single machine. That works just fine, but the relation between the services often get lost in the noise. This is unhelpful for new users trying to understand Plan 9. Also, when installing a single Plan 9 machine with all services, it is hard to administer the machine: almost everything is different in Plan 9, and new users won't even know how to edit files.

about this guide

This guide describes how you can install a Plan 9 network on an OpenBSD machine (it will probably work on any unix machine though). The authentication service (called "authsrv" on Plan 9) is provided by a unix version: authsrv9. The file service is provided by a program called "u9fs". It comes with Plan 9. Both run from inetd. The (diskless) cpu server is provided by running qemu, booted from only a floppy (so without local storage). Finally, the terminal is provided by the program drawterm. The nice thing about this approach is that you can use all your familiar unix tools to get started with Plan 9 (e.g. you can edit the Plan 9 files with your favorite unix editor). I'm assuming you have read at least something about Plan 9, for example the introduction paper Plan 9 from Bell Labs.

help

If you need help, you can search the archives of the 9fans mailing list, consult the Plan 9 wiki, read the Plan 9 manual pages, or read some papers on Plan 9. If you have problems specific to this guide (or tips for improvement), you can contact me at mechiel@ueber.net. If you have generic Plan 9 problems, you can ask on the irc channel #plan9 on irc.freenode.net, or send mail to 9fans, the Plan 9 mailing list.

your network configuration

This guide configures the services to be accessible over the local ethernet. Each service gets a different ip address. This makes it easy to replace parts of your network by different (physical) machines later on. Now is a good moment to think about the ip addresses for your Plan 9 network. Mine is fairly typical: a 192.168.1.0/24 network. I chose the following ip addresses and host names for the services:

```
192.168.1.188  ufs1.local
192.168.1.189  auth1.local
192.168.1.193  cpu1.local
```

file server

intro

A Plan 9 file server provides its service over TCP, talking the 9P2000 protocol (often called just "9p"). The protocol has requests for opening, reading, writing, stat'ing, closing files, etc. On a real Plan 9 system, the program delivering the file service is called fossil. Fossil stores the files on local storage (hard drives), in its file system format. We will use u9fs, which just uses the unix file system to store its files.

creating users for u9fs

So let's start with configuring u9fs. First we create some unix users to own the files. U9fs needs four users on the file server: bootes, sys, adm and none. Bootes is the user a cpu server starts as (it does not own files). Sys owns the system files (all the binaries, the source code, the man pages, etc.). Adm owns some configuration files and sensitive files such as those containing passwords. None is the equivalent of unix's nobody. Create these users (e.g. with adduser(8)), putting them in their own groups. Give user sys and adm the umask 002 (in their .profile), so all files they create are group-writeable and world-readable.

plan 9 files

Next, we need a copy of the files for a Plan 9 system. Download the iso from http://plan9.bell-labs.com/plan9/download.html. Find a spot on your file system for the Plan 9 install to live. I chose /usr/ufs1 (my file server is called ufs1). Now mount the iso, and copy its contents to /usr/ufs1/. For example:

```
# as root
vnconfig svnd0 /path/to/plan9.iso
mount /dev/svnd0c /mnt
cp -R /mnt /usr/ufs1

# as root, now fix up the owners & permissions
# this reads the file the plan 9 installer uses to determine which files
to copy,
# turns them into unix chown & chmod commands,
# replaces two users (upas & glenda) we don't have by user sys,
# and finally executes the commands by the unix shell.
cd /usr/ufs1
cat dist/replica/plan9.log | \
        sed -n 's/[0-9][0-9]* [0-9][0-9]* [am] \([^ ]*\) - .*\([0-9]
[0-9][0-9]\) \([a-zA-Z][a-zA-Z]*\) \([a-zA-Z][a-zA-Z]*\) \([0-9][0-9]*\)
[0-9][0-9]*/chown \3:\4 \1; chmod \2 \1; touch -t `date -r \5 +%Y%m%d%H
%M.%S` \1/p' | \
        sed 's/^chown glenda:glenda/chown sys:sys/' | \
        sed 's/^chown upas:upas/chown sys:sys/' | \
        sh
```

NOTE: executing the command above will give a bunch of error messages about files not existing. These files all end with a colon (Plan 9 gives these files different names on the iso), or have utf-8 in their names (OpenBSD's cd9660 file system code does not understand unicode). The files are not important for now, but you will want to fix them later on. The appendix on updating your system has instructions on how fetch those files.

compiling u9fs

You need a u9fs binary. The source comes with Plan 9, so the easiest method is to compile it from the Plan 9 files you just copied to /usr/ufs1:

```
# as user sys:
cd /usr/ufs1/sys/src/cmd/unix/u9fs
make LDFLAGS=-static  # static because we chroot it.

# as root:
mkdir -m700 /usr/ufs1/unix
install -m700 -o root -g wheel /usr/ufs1/sys/src/cmd/unix/u9fs/u9fs
/usr/ufs1/unix/u9fs
```

users on u9fs

The file server does not explicitly have to know who valid users are. It will just hear from the authentication server that some user is valid (through an authentication ticket presented by the client), and u9fs will try to give it access: it will change the uid from root to that user. In Plan 9 users are identified by name, not by numeric id. U9fs converts Plan 9 user names to unix uid by consulting the /etc/passwd file (or rather, /etc/pwd.db), and /etc/groups. Since we chroot u9fs, we need to populate /usr/ufs1/etc with those files:

```
# as root
mkdir -m711 /usr/ufs1/etc
install -m644 /etc/passwd /etc/pwd.db /etc/group /usr/ufs1/etc
```

Later on, when I log in as Plan 9 user "mjl", u9fs will consult the password database, find the numeric uid, and change the processes uid (thus dropping root priviliges). Now only normal unix file system permissions restrict access to files served by u9fs. Most files (e.g. all system binaries, manual pages and sources) are world-readable, so as user mjl I can read those just fine. I cannot write them though (since I am not a member of unix group sys). Missing so far is a home directory for user mjl. Let's create it:

```
# as root
mkdir -m775 /usr/ufs1/usr/mjl
chown mjl:mjl /usr/ufs1/usr/mjl
```

starting u9fs

Next we need to start the u9fs service. As mentioned before, it runs from inetd. Add the following line to /etc/inetd.conf:

```
192.168.1.188:564  stream  tcp  nowait  root  /usr/sbin/chroot chroot
/usr/ufs1 /unix/u9fs -a p9any -A /unix/u9fs.key -l /unix/u9fs.log
```

U9fs listens only on its own ip address, on port 564 (the called 9fs in Plan 9), it chroots to /usr/ufs1 but keeps running as root at first. The executed binary is in /usr/ufs1/unix/ufs9. Option "-l /unix/u9fs.log" specifies the log file (handy for debugging, option -D improves verbosity). Options -a and -A are for authentication. The authentication method is "p9any", and the authentication credentials are in /unix/u9fs.key. We'll set up the authentication server in the next secion, so we'll come back to /unix/u9fs.key then. For now, we'll just create an empty file, and one for the log file too:

```
# as root
echo -n >/usr/ufs1/unix/u9fs.key
echo -n >/usr/ufs1/unix/u9fs.log
chmod 600 /usr/ufs1/unix/u9fs.key /usr/ufs1/unix/u9fs.log
```

You can "pkill -HUP inetd" now to activate the service. Except for the authentication info, the file server has been configured.

authentication server

authsrv & authsrv9

Now we'll configure the authentication server. In Plan 9 this program is called authsrv. It is just a TCP service started from listen(8), the Plan 9-equivalent of inetd. Authsrv needs access to users encryption keys (derived from their passwords). These keys are stored safely, and made available only to authsrv using a Plan 9 program called keyfs. We'll use a program similar to authsrv but runs on unix: authsrv9. We won't use a keyfs-equivalent: authsrv9 solely relies on unix file permissions to keep the keys safe. You can get authsrv9 here: http://www.ueber.net/code/r/authsrv9. It helps to need to know a little bit about authentication in Plan 9.

about authentication on plan 9

The authentication server knows the keys of all users in its authentication domain (that's why they are specially protected by keyfs on Plan 9). When a user (the client) wants to login to a service (e.g., a file server), the client first connects to the file server. The file server tells the client who the "owner" of the file server is (called the "authentication id", or just "authid") and to which "authentication domain" ("authdom") the service belongs. The authid is just a username, usually "bootes" by convention. The authdom is often a DNS domain name, but does not have to be. For clients, the authdom essentially just allows it to determine which authentication server it needs to get tickets from. The configuration file /lib/ndb/local (explained later) contains the mapping from authdom to authentication server addresses. The client now connects to the authentication server, and ask for a pair of tickets: one for his own username, for logging into the machine with the authid (e.g. "bootes"). The client then uses these tickets in a protocol with the server to let both sides establish that they are who they claim to be (the protocol is called "p9any"). To make this work, any server in your Plan 9 network needs to be configured with the same authid, the same authdom, and the same authentication key (derived from a password). This is important: the authentication server, the file server, and the cpu server will all be configured with the same authentication id, domain & key. In this guide, authid is "bootes", authdom is "local" and the key is secret.

configuring authsrv9

Now we are ready to configure authsrv9. Get a copy, type "make" to compile it. We need to find a place to store the configuration and user keys. I chose /var/authsrv9. We'll chroot to that location again, and make the service run as user _authsrv9 (you need to create this user).

```
# as root
mkdir /var/authsrv9
mkdir /var/authsrv9/dev  # for /dev/log, for syslog
mkdir /var/authsrv9/bin  # for the binary
mkdir /var/authsrv9/auth # holding all configuration files
mkdir /var/authsrv9/auth/users  # holding info about users
```

```
# we'll fix permissions further down
```

config files for authsrv9

The manual page for authsrv9 explains the meaning of the files in /var/authsrv9/auth. This configures them:

```
# as root
echo -n bootes >/var/authsrv9/auth/authid
echo -n local >/var/authsrv9/auth/authdom
echo adm >/var/authsrv9/auth/badusers
```

passwords & keys

Next we create the users "bootes", "sys" and your username (mine is "mjl"). Keys are created by the passtokey program that comes with authsrv9. It derives a 7-byte DES key from the password typed in (must be >=8 and <=28 bytes).

```
# as root
mkdir /var/authsrv9/auth/users/bootes
echo -n ok >/var/authsrv9/auth/users/bootes/status  # "disabled" (and
anything else) disables the user
echo -n never >/var/authsrv9/auth/users/bootes/expire  # alternatively,
write a unix timestamp

mkdir /var/authsrv9/auth/users/mjl
echo -n ok >/var/authsrv9/auth/users/mjl/status
echo -n never >/var/authsrv9/auth/users/mjl/expire

mkdir /var/authsrv9/auth/users/sys
echo -n ok >/var/authsrv9/auth/users/sys/status
echo -n never >/var/authsrv9/auth/users/sys/expire

/path/to/passtokey >/var/authsrv9/auth/users/mjl/key
/path/to/passtokey >/var/authsrv9/auth/users/sys/key
/path/to/passtokey >/var/authsrv9/auth/users/bootes/key
```

And now for the permissions:

```
# as root

# all files are owned by root:wheel now, authsrv9 needs read access
chgrp -R _authsrv9 /var/authsrv9
```

```
chmod -R u=rwX,g=rX,o= /var/authsrv9

# authsrv9 needs write access to the keys:
# users can change their passwords and a new key must be written
chmod g+w /var/authsrv9/auth/users/*/key
```

authid, authdom & authkey

The password for bootes is used for the authentication key of your Plan 9 network. This is the key/password that must be the same on all servers in your network. Now that you have chosen the password for bootes, we can go back to u9fs and configure that last missing piece. The file /usr/ufs1 /etc/u9fs.key must contain three lines: the auth password, auth id, auth domain. NOTE: the first line contains the plain text *password*, not the key. The second line will contain "bootes". The third line contains "local".

chroot & syslog

Authsrv9 is now configured, and we just need to start it. First, restart syslogd with the additional option "-a /var/authsrv9/dev/log", to get an extra syslog socket, for the chroot environment. Make it permanent by adding this to /etc/rc.conf.local's syslogd_flags, e.g.:

```
syslogd_flags="-a /var/authsrv9/dev/log"
```

starting authsrv9

Next, copy the authsrv9 binary to /var/authsrv9/bin/authsrv9 (it's statically linked, so it will just work).

```
# as root, in authsrv9 source directory
install -m750 -o root -g _authsrv9 authsrv9 /var/authsrv9/bin/authsrv9
```

Then add these lines to inetd, and pkill -HUP inetd. As you can see the service listens only on its own ip address, on port 567, called "ticket" on Plan 9.

```
192.168.1.189:567  stream  tcp  nowait  root  /usr/sbin/chroot chroot
/var/authsrv9 /bin/authsrv9
```

The authentication service has been configured.

cpu server

intro

Next is the cpu server. The cpu server provides the possibility to run Plan 9 processes. Thus we really need to run a Plan 9 kernel for this. We'll use qemu as computer emulator. As explained before, the cpu server does not have local storage, all files are served by the file server. For network, we'll use qemu's tap network support. The default OpenBSD qemu tap configuration creates a bridge that contains both the tap device and your ethernet device: exactly what we need. You need to set environment variable ETHER to your network interface, and add options "-net nic,macaddr=XX:XX:XX:XX:XX:XX -net tap" to the qemu command-line. I chose a mac address similar to one used as example in the manual page.

To make your cpu server useful after it booted, be sure to setup DHCP so the cpu server will get a lease (preferably with a fixed ip address). Alternatively, you could skip DHCP and configure the boot image with a fixed ip addres, but that is outside the scope of this document.

kernel & boot image

We are now missing one big thing: the kernel to boot. I prepared a floppy image that will boot a cpu server kernel, get the image here: files/floppycpu1.img.gz. Gunzip the image after downloading, and use it on the qemu command-line with option `-fda`. Instructions on how to create your own floppy image can be found in appendix b.

Next we add one more thing for convenience: another floppy where the authid, authdom and authentication key can be stored. We can choose not to store this at all, but then you would have to type it in each time you boot the cpu server. Alternatively, you could store it on the same floppy as the kernel is on. But then you'll have to go through more trouble to update the kernel on the disk image (instead, we can just replace the floppy with the kernel while keeping the floppy with the authentication information). Hence, just for convenience, we add another floppy image. Only a few sectors are used on it, so the compressed version is very small. Get it here: files/floppynvr.img.gz. During first boot, it will detect that the authentication config on the floppy isn't valid yet, prompt for a new config (authid, etc) and write it to the floppy (so on subsequent boots it will use this data).

qemu invocation

Finally, this is the complete command-line (I added "-name cpu1" to have a useful title in my window manager):

```
sudo ETHER=em0 qemu -m 256 -net nic,macaddr=52:54:00:12:34:56 -net tap
-fda floppycpu1.img -fdb floppynvr.img -name cpu1
```

network configuration

Before we boot the cpu server, we need to configure a few more things. First, we want every process on the cpu server (including terminals that log in) to have access to our Plan 9 network configuration. We do that by adding this info to /lib/ndb/local:

```
# let programs determine that auth1.local is the authentication server.
auth=auth1.local authdom=local
```

```
# specify ip addresses for host names
sys=ufs1 dom=ufs1.local ip=192.168.1.188
sys=auth1 dom=auth1.local ip=192.168.1.189
sys=cpu1 dom=cpu1.local ip=192.168.1.193

# let programs on the cpu server find the default cpu/auth/fs server for
the network
ipnet=localnet ip=192.168.1.0 ipmask=255.255.255.0
        auth=auth1
        fs=ufs1
        cpu=cpu1
```

The lines in this file contain key-value pair, one or more per line. Programs can look for lines matching some key-value pair. For example, when the program performing authentication (called factotum) learns that a file server is in the authentication domain "local" it will look for the "auth" attribute in lines matching "authdom=local"). The indented line is a form of continuation.

booting the cpu server

It is time to boot the cpu server. It will first ask for two ip addresses: one for the authentication server and one for the file server. Type them in.

Next, the cpu server detects that the authentication configuration is not valid. So it asks for the authid ("bootes"), authdom ("local"), sectore password (we won't configure that here, so just hit enter to skip it), and the authentication password (the password for bootes).

You will now see a black screen with a prompt. Congratulations, your Plan 9 network has been set up. Next is logging into it from a terminal.

terminal

drawterm

The network has been set up: we have an authentication, file & cpu server. We can now use a terminal to use the network, we'll use drawterm. Drawterm is really a unix program that does most of what a normal Plan 9 terminal would do. But instead of running on bare hardware, it runs as a program on unix. (Alternatively, you could boot a qemu that will act as a terminal; but let's keep it simple for now). Get a copy of drawterm from http://swtch.com/drawterm/drawterm.tgz. Type `CONF=unix make AUDIO=none` to get the drawterm binary. Now to log in as Plan 9 user mjl:

```
drawterm -a 192.168.1.189 -c 192.168.1.193 -u mjl
```

The argument to `-a` is the authentication server (ip address or host name), `-c` specifies the cpu server. Starting this will pop up an X11 window asking for the password for `mjl@local` (i.e. user "mjl" at authentication domain "local"). Type your password, and you will be presented with a shell prompt. To populate your home directory with the basic files, run /sys/lib/newuser. Among others, this will give you a lib/profile (the equivalent of .profile on unix), and then execute lib/profile. This will start rio (the window manager) which will show just a gray screen. Click and hold the right mouse button and release when on "New" to create a new window. Click and hold the right button again and draw the contours of the new window, release to create the window. The new window runs a shell from which you can start any program. A nice and quick introduction to using the Plan 9 user interface, and some of the programs, is A Plan 9 Newbie's Guide.

conclusion

working plan 9 network

You should now have a running Plan 9 network. You should be able to try out Plan 9 programs (the user interface: acme & rio, use the authentication agent factotum, play around with 9p and the everything-is-a-file(-server) paradigm, learn how to keep your system updated, etc.). Hopefully, you also have a mental picture of how the authentication, file, cpu server and the terminal are all separate entities that talk to each other over the network.

this is just the beginning...

We did cheat a bit: We only used a real Plan 9 kernel on the cpu server. And even there we just used qemu, not even real hardware. But you can now easily experiment with and expand your Plan 9 network. For example, you could add a cpu server running on real hardware. Or, you could use real hardware for the terminal. Both can be done without using local storage on the machine, so you can try it non-destructively (e.g. by booting a Plan 9 kernel using PXE, or booting the plan9.iso from a CD). More intrusive, but also very worthwhile, is installing the Plan 9 file server fossil (which goes hand in hand with venti). The Plan 9 file server has the interesting feature of persistent snapshots. U9fs itself is not perfect: it does not support the append-only and exclusive-use mode bit on files (though a hack to support those is on its way). An earlier file server (but not necessarily worse, it does snapshots too!) is available as well. Alternatively, with your new understanding of Plan 9, you could install a single Plan 9 machine that is auth, fs and cpu server in one. There is enough room for playing with Plan 9 and improving your network.

appendix a: making a cpu floppy image

Making a cpu server floppy boot image, as used to boot the diskless cpu server, is not hard once you have a cpu server up and running. The boot floppy contains three files: the kernel, a boot loader and a configuration file called plan9.ini. The bootloader already come with Plan 9: /386/9load. The kernel can be compiled as follows:

```
# from drawterm, logged as user sys
```

```
cd /sys/src/9/pc
ramfs

# create a kernel config similar to pccpuf, but comment-out fossil & venti
(they are big and make the floppy too big)
cat pccpuf | sed 's,^.*fossil/fossil.*,#&,' | sed 's,^.*venti/venti.*,#&,'
>pccpufu

mk 'CONF=pccpufu'
gzip <9pccpufu >9pccpufu.gz
```

The configuration file contains the location of the kernel on the floppy, vga configuration (resolution) and some other options. The following commands (requiring no special privileges) create the image /tmp/floppycpu1.img:

```
cat >/tmp/plan9.ini<<EOF
bootfile=fd0!9pccpufu.gz
bootargs=tcp
nobootprompt=tcp
sysname=cpu1
# fs=192.168.1.188
# auth=192.168.1.189

#*noahciload=1
*debugload=1
*nobiosload=1
*nodumpstack=1
*nomp=1
dmamode=ask
partition=new
mouseport=ps2
monitor=xga
vgasize=800x600x8
#apm0=
EOF
echo -n >/tmp/floppycpu1.img
disk/format -b /386/pbs -df /tmp/floppycpu1.img /386/9load /tmp/plan9.ini
/sys/src/9/pc/9pccpufu.gz
```

You can now boot qemu from the resulting file /tmp/floppycpu1.img.

NOTE: the sysname= attribute specifies the system name the cpu server uses. Change it here if you named your cpu server differently.

NOTE: the fs= and auth= attributes specify the ip addresses of the file server and authentication server. If you don't want to type them each time you boot, you can change them here and create a new floppy boot image.

Creating the second floppy that holds the authentication information is even easier. It is just a floppy with a FAT partition that contains a 1-sector file called plan9.nvr. Create it as follows:

```
echo -n >/tmp/floppynvr.img
dd -if /dev/zero -of /tmp/plan9.nvr -count 1  # block size is 512 bytes by
default
disk/format -df /tmp/floppynvr.img /tmp/plan9.nvr
```

appendix b: more configuration

There are a few more things you can configure on a Plan 9 system. We skipped them in this guide because it would have made it far too large. Here are a few tasks you might want to perform:

setting the time zone

A cpu server reads its timezone from the file /adm/timezone/local. Many timezones are available to choose from in /adm/timezone. Just copy the file for the time zone you are in the /adm/timezone/local (as user adm, from unix) and you are done.

start rio on cpu server

When you boot your cpu server, you'll see a black screen with a prompt. By now you have hopefully played around with rio, the window manager, and perhaps acme or sam (editors) too. You can configure the cpu server to start rio at startup. It is pretty simple. Just make the file /cfg/cpu1/cpustart (or replace cpu1 with your cpu servers name). Make it executable and put the following in it:

```
#!/bin/rc

for(i in m i S t)
        bind -a '#'^$i /dev

aux/mouse $mouseport
aux/vga -l $vgasize
rio
```

At boot time, the cpu server will run the script /usr/ufs1/rc/bin/cpurc, which will execute /cfg/$sysname /cpustart (where $sysname is the name of your system). Sysname was configured in the floppy boot image, in the file plan9.ini. The resolution of the display was also configured in plan9.ini.

starting fewer services

Cpu servers start many of network services by default (identd, ftpd, smtpd, pop3, imap4d, etc.). Due to limitations of authsrv9, most of these services cannot authenticate Plan 9 users. And I was not going to use them anyway. So I disabled them. To do the same, just make only the ones you want to keep executable:

```
# on unix, as user sys
chmod -x /usr/ufs1/rc/bin/service/*
chmod +x /usr/ufs1/rc/bin/service/tcp17007  # allow import(4) to work
chmod +x /usr/ufs1/rc/bin/service/tcp17010  # allow cpu(1) to work (which
is what drawterm does)
```

appendix c: updating

Updating the system is done using the "replica" tools. These download changes from Bell Labs and apply them to your system. Files can be deleted, added, or modified, but care is taken so that local changes are not overwritten. Since we do not have a Plan 9 file such as fossil, we need to configure the updating tools a bit. First, create /usr/ufs1/dist/replica/unetwork (with same permissions as /usr/ufs1/dist/replica/network):

```
#!/bin/rc

if(~ $replica '')
	replica=replica

s=/n/dist/dist/replica
serverroot=/n/dist
serverlog=$s/plan9.log
serverproto=$s/plan9.proto
fn servermount {
	9fs sources
	bind /n/sources/plan9 /n/dist
}
fn serverupdate { status='' }

fn clientmount { 9fs 'net!$fs!9fs' }
c='/n/$fs!9fs/dist/replica'
clientroot='/n/$fs!9fs'

clientdb=$c/client/plan9.db
clientexclude=(dist/replica/client)
clientlog=$c/client/plan9.log
clientproto=$c/plan9.proto
```

```
applyopt=(-u -T$c/client/plan9.time)
```

Next, copy the state of the system from the installer to what our local system represents and fix permissions:

```
# as root, on unix
mkdir /usr/ufs1/dist/replica/client
cp /usr/ufs1/dist/replica/plan9.db /usr/ufs1/dist/replica/client/plan9.db
chmod u+w /usr/ufs1/dist/replica/client/plan9.db
chown -R sys /usr/ufs1/dist/replica
```

Now you are set. Login as user sys using drawterm, start a rio, open a new window, start ramfs (for temporary files) and execute the replica/pull program:

```
drawterm -a 192.168.1.189 -c 192.168.1.193 -u sys

bind /mnt/term/dev/cons /dev/cons
bind /mnt/term/dev/consctl /dev/consctl
bind -a /mnt/term/dev /dev
rio

# open window
ramfs
replica/pull -v /dist/replica/unetwork
```

NOTE: the method described above has a drawback: you can only write files as user sys. Most files are owned by sys, so for those it isn't a problem. But if some files owned by adm are updated, the owner of the file cannot be set correctly. You must manually fix those. Normal Plan 9 does not have this limitation: It disables permission checking for the program that updates the files.

getting the missing files

Finally, we were missing a few files (those containing colons, and those with utf-8 names). The easiest way to fix this is just to copy the files from "sources" (the Bell Labs server containing the latest files). You can mount sources by `9fs sources`. A copy of the latest Plan 9 files are then available in /n/sources/plan9. When you copy the files to ufs1, get a clean copy of the file server first by executing `9fs ufs1`. Now copy the missing files from /n/sources/plan9/... to /n/ufs1/.... The missing files are:

```
# directories
n/a:
n/c:
n/d:
sys/doc/8½
```

```
# files
rc/bin/9fat:
rc/bin/a:
rc/bin/b:
rc/bin/c:
rc/bin/usbfat:
sys/doc/8½/8½.ms
sys/doc/8½/8½.ps
sys/doc/8½/fig1.ps
sys/doc/8½/mkfile
sys/lib/dist/pc/sub/a:
sys/lib/dist/pc/sub/boota:
sys/lib/troff/font/devutf/charlib/☺
sys/src/9/bitsy/devμc.c
```