Prologin System Administration

Release 2020

Association Prologin

Aug 16, 2020

CONTENTS

This documentation hopefully explains everything there is to know about the infrastructure used at Prologin to host the finals of the contest. Our needs are surprisingly complex to meet with our low budgets and low control over the hardware and network, which explains why some things seem very complicated.

CHAPTER

ONE

INFRASTRUCTURE OVERVIEW

This section describes what runs on our servers and what it is used for.

1.1 Needs

- Host 100 contest participants + 20 organizers on diskless computers connected to a strangely wired network (2 rooms with low bandwidth between the two).
- **Run several internal services:**
  - DHCP + DNS
  - Machine DataBase (MDB)
  - User DataBase (UDB)
  - NTPd
- **Run several external services (all of these are described later):**
  - File storage
  - Homepage server
  - Wiki
  - Contest website
  - Bug tracking software (Redmine)
  - Documentation pages
  - IRC server
  - Pastebin
  - Matches cluster

1.2 Network infrastructure

We basically have two local networks:

- User LAN, containing every user machine (Pasteur + IP12A) and all servers.
- Matches LAN, containing the cluster master and all the cluster slaves.

The User LAN uses 192.168.0.0/24, and the gateway (named `gw`) is 192.168.1.254. 192.168.1.0/24 is reserved for servers, and 192.168.250.0/24 is reserved for machines not in the MDB.

The Matches LAN uses 192.168.2.0/24, and the gateway (named `gw.cl`) is 192.168.2.254.

Both `gw` and `gw.cl` communicate through an OpenVPN point to point connection.

1.3 Machine database

The Machine DataBase (MDB) is one of the most important part of the architecture. Its goal is to track the state of all the machines on the network and provide information about the machines to anyone who needs it. It is running on `mdb` and exports a web interface for administration (accessible to all roots).

A Python client is available for scripts that need to query it, as well as a very simple HTTP interface for use in PXE scripts.

It stores the following information for each machine:

- Main hostname
- Alias hostnames (mostly for services that have several DNS names)
- IP
- MAC
- Nearest root NFS server
- Nearest home NFS server
- Machine type (user, orga, cluster, service)
- Room id (pasteur, alt, cluster, other)

It is the main data source for DHCP, DNS, monitoring and other stuff.

When a machine boots, an IPXE script will lookup the machine info from the MDB (to get the hostname and the nearest NFS root). If it is not present, it will ask for information on stdin and register the machine in the MDB.

1.4 User database

The User DataBase (UDB) stores the user information. As with MDB, it provides a simple Python client library as well as a web interface (accessible to all organizers, not only roots). It is running on `udb`.

It stores the following information for every user:

- Login
- First name
- Last name
- Current machine name

- Password (unencrypted so organizers can give it back to people who lose it)
- At his computer right now (timestamp of last activity)
- Type (contestant, organizer, root)
- SSH key (mostly useful for roots)

As with the MDB, the UDB is used as the main data source for several services: every service accepting logins from users synchronizes the user data from the UDB (contest website, bug tracker, …). A `pam_udb` service is also used to handle login on user machines.

1.5 File storage

4 classes of file storage, all using NFS over TCP (to handle network congestion gracefully):

- Root filesystem for the user machines: 99% reads, writes only done by roots.
- Home directories filesystem: 50% reads, 50% writes, needs low latency
- Shared directory for users junk: best effort, does not need to be fast, if people complain, tell them off.
- Shared directory for champions/logs/maps/…: 35% reads, 65% writes, can't be sharded, needs high bandwidth and low latency

Root filesystem is manually replicated to several file servers after any change by a sysadmin. Each machine using the root filesystem will interogate the MDB at boot time (from an IPXE script) to know what file server to connect to. These file servers are named `rfs-1, rfs-2, ...`. One of these file servers (usually `rfs-1`) is aliased to `rfs`. It is the one roots should connect to in order to write to the exported filesystem. The other rfs servers have the exported filesystem mounted as read-only, except when syncing.

Home directories are sharded to several file servers. Machines interogate the MDB to know what home file server is the nearest. When a PAM session is opened, a script interogates the UDB to know what file server the home directory is hosted on. If it is not the correct one, it sends a sync query to the old file server to copy the user data to the new file server. These file servers are named `hfs-1, hfs-2, ...`

The user shared directory is just one shared NFS mountpoint for everyone. It does not have any hard performance requirement. If it really is too slow, it can be sharded as well (users will see two shared mount points and will have to choose which one to use). This file server is called `shfs`.

The shared directory for matches runners is not exposed publicly and only machines from the matches cluster can connect to it. It is a single NFS mounpoint local to the rack containing the matches cluster. The server is connected with 2Gbps to a switch, and each machine from the cluster is connecter do the same switch with a 1Gbps link. This file server is running on `fs.cl`, which is usually the same machine as `gw.cl`.

1.6 DHCP and DNS

The DHCP server for the user network runs on `gw`. It is responsible for handing out IPs to machines connecting to the network. The MAC<->IP mapping is generated from MDB every minute. Machines that are not in the MDB are given an IP from the 192.168.250.0/24 range.

The DHCP server for the cluster network runs on `gw.cl`. The MAC<->IP mapping is also generated from MDB, but this time the unknown range is 192.168.2.200 to 192.168.2.250.

The DNS server for the whole infrastructure runs on `ns`, which is usually the same machine as `gw`. The hostname<->IP mapping is generated from MDB every minute. There are also some static mappings for the unknown ranges: 192.168.250.x is mapped to `alien-x` and 192.168.2.200-250 is mapped to `alien-x.cl`.

1.7 Matches cluster

The matches cluster contains several machines dedicated to running Stechec matches. It is a separate physical architecture, in a separate building, on a separate LAN. The two gateways, `gw.cl` and `gw` are connected through an OpenVPN tunnel.

`master.cl` runs the Stechec master node, which takes orders from the Stechec website (running on `contest`, on the main LAN). All nodes in the cluster are connected to the master node.

To share data, all the nodes are connected to a local NFS share: `fs.cl`. Read the file storage overview for more information.

1.8 Other small services

Here is a list of all the other small services we provide that don't really warrant a long explanation:

- Homepage: runs on `homepage`, provides the default web page displayed to contestants in their browser
- Wiki: runs on `wiki`, UDB aware wiki for contestants
- Contest website: runs on `contest`, contestants upload their code and launch matches there
- Bug tracker: `bugs`, UDB aware Redmine
- Documentations: `docs`, language and libraries docs, also rules, API and Stechec docs.
- IRC server: `irc`, small UnrealIRCd without services, not UDB aware
- Paste: `paste`, random pastebin service

CHAPTER

TWO

SETUP INSTRUCTIONS

If you are like the typical Prologin organizer, you're probably reading this documentation one day before the start of the event, worried about your ability to make everything work before the contest starts. Fear not! This section of the documentation explains everything you need to do to set up the infrastructure for the finals, assuming all the machines are already physically present. Just follow the guide!

Maintainers:

- Alexandre Macabies (2013-2019)
- Antoine Pietri (2013-2018)
- Rémi Audebert (2014-2019)
- Paul Hervot (2014, 2015)
- Marin Hannache (2013, 2014)
- Pierre Bourdon (2013, 2014)
- Nicolas Hureau (2013)
- Pierre-Marie de Rodat (2013)
- Sylvain Laurent (2013)

2.1 Step 0: hardware and network setup

Before installing servers, we need to make sure all the machines are connected to the network properly. Here are the major points you need to be careful about:

- Make sure to balance the number of machines connected per switch: the least machines connected to a switch, the better performance you'll get.
- Inter-switch connections is not very important: we tried to make most things local to a switch (RFS + HFS should each be local, the rest is mainly HTTP connections to services).
- Have a very limited trust on the hardware that is given to you, and if possible reset them to a factory default.

For each pair of switches, you will need one RHFS server (connected to the 2 switches via 2 separate NICs, and hosting the RFS + HFS for the machines on these 2 switches). Please be careful out the disk space: assume that each RHFS has about 100GB usable for HFS storage. That means at most 50 contestants (2GB quota) or 20 organizers (5GB quota) per RHFS. With contestants that should not be a problem, but try to balance organizers machines as much as possible.

You also need one gateway/router machine, which will have 3 different IP addresses for the 3 logical subnets used during the finals:

Users and services 192.168.0.0/23

Alien (unknown) 192.168.250.0/24

Upstream Based on the IP used by the bocal internet gateway.

Contestants and organizers must be on the same subnet in order for UDP broadcasting to work between them. This is required for most video games played during the finals: server browsers work by sending UDP broadcast announcements.

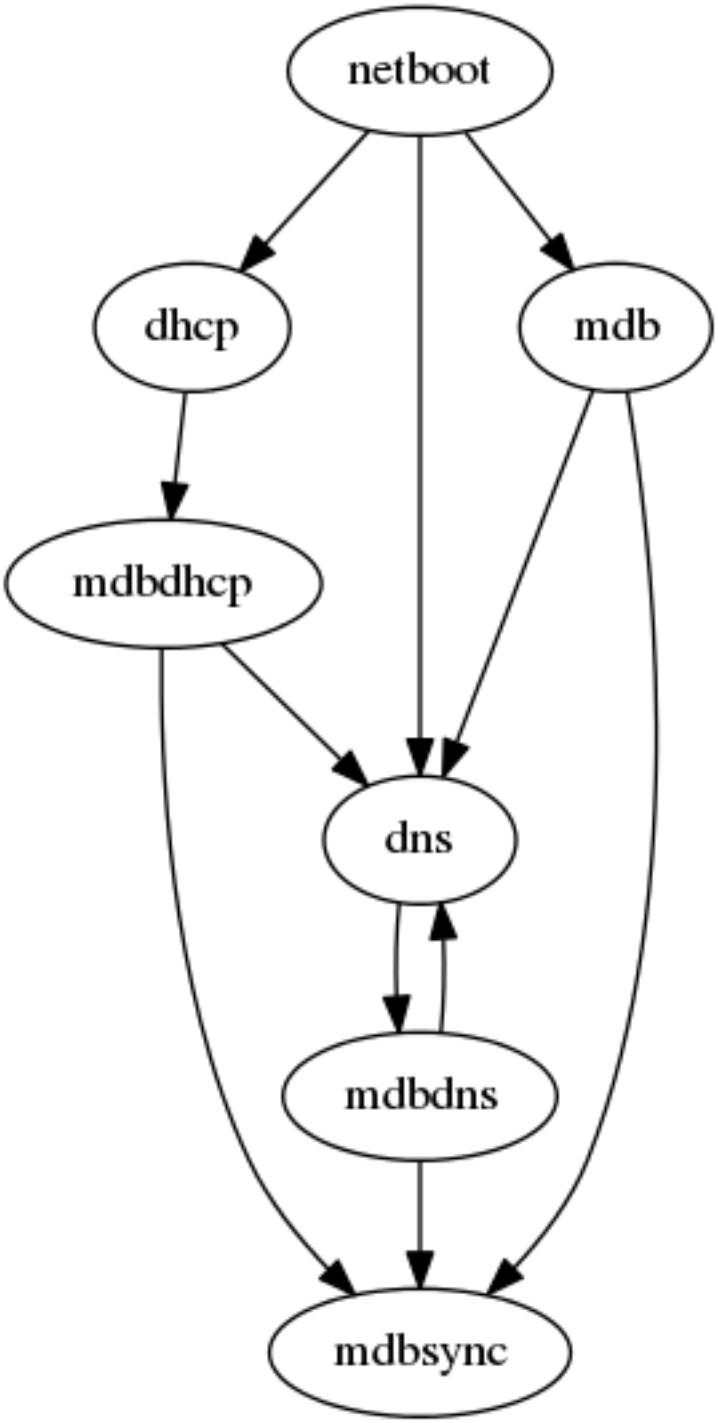
Having services and users on the same logical network avoids all the traffic from users to services going through the gateway. Since this includes all RHFS traffic, we need to make sure this is local to the switch and not being routed via the gateway. However, for clarity reasons, we allocate IP addresses in the users and services subnet like this:

Users 192.168.0.0 - 192.168.0.253

Services and organizers machines 192.168.1.0 - 192.168.1.253

2.2 Step 1: setting up the core services: MDB, DNS, DHCP

This is the first and trickiest part of the setup. As this is the core of the architecture, everything kind of depends on each other:



Fortunately, we can easily work around these dependencies in the beginning.

All these core services will be running on `gw`, the network gateway. They could run elsewhere but we don't have a lot of free machines and the core is easier to set up at one single place.

The very first step is to install an Arch Linux system for `gw`. We have scripts to make this task fast and easy.

2.2.1 Basic system: file system setup

Note: The installation process is partially automated with scripts. You are strongly advised to read them and make sure you understand what they are doing.

Let's start with the hardware setup. You can skip this section if you are doing a containerized install or if you already have a file system ready.

For `gw` and other critical systems such as `web`, we setup a RAID1 (mirroring) over two discs. Because the RAID will be the size of the smallest disc, they have to be of the same capacity. We use regular 500GBytes SATA, which is usually more than enough. It is a good idea to choose two different disks (brand, age, batch) to reduce the chance to have them failing at the same time.

On top of the RAID1, our standard setup uses LVM to create and manage the system partition. For bootloading the system we use the good old BIOS and `syslinux`.

All this setup is automated by our bootstrap scripts, but to run them you will need a bootstrap Linux distribution. The easiest solution is to boot on the Arch Linux's install medium https://wiki.archlinux.org/index.php/Installation_guide#Boot_the_live_environment.

Once the bootstrap system is started, you can start the install using:

```
bash <(curl https://raw.githubusercontent.com/prologin/sadm/master/install_scripts/
↪bootstrap_from_install_medium.sh)
```

This script checks out sadm, then does the RAID1 setup, installs Arch Linux and configures it for RAID1 boot. So far nothing is specific to sadm and you could almost use this script to install yourself an Arch Linux.

When the script finishes the system is configured and bootable, you can restart the machine:

```
reboot
```

The machine should reboot and display the login tty. To test this step:

- The system must boot
- Systemd should start without any `[FAILED]` item.
- Log into the machine as `root` with the password you configured.
- Check that the hostname is `gw.prolo` by invoking `hostnamectl`:

```
 Static hostname: gw.prolo
       Icon name: computer-container
         Chassis: container
      Machine ID: 603218907b0f49a696e6363323cb1833
         Boot ID: 65c57ca80edc464bb83295ccc4014ef6
  Virtualization: systemd-nspawn
Operating System: Arch Linux
          Kernel: Linux 4.6.2-1-ARCH
    Architecture: x86-64
```

- Check that the timezone is `Europe/Paris` and NTP is enabled using `timedatectl`:

```
      Local time: Fri 2016-06-24 08:53:03 CEST
  Universal time: Fri 2016-06-24 06:53:03 UTC
        RTC time: n/a
       Time zone: Europe/Paris (CEST, +0200)
```

(continues on next page)

(continued from previous page)

```
 Network time on: yes
NTP synchronized: yes
 RTC in local TZ: no
```

- Check the NTP server used:

```
systemctl status systemd-timesyncd
Sep 25 13:49:28 halfr-thinkpad-e545 systemd-timesyncd[13554]: Synchronized to␣
↪time server 212.47.239.163:123 (0.arch.pool.ntp.org).
```

- Check that the locale is `en_US.UTF8` with the `UTF8` charset using `localectl`:

```
System Locale: LANG=en_US.UTF-8
    VC Keymap: n/a
   X11 Layout: n/a
```

- You should get an IP from DHCP if you are on a network that has such a setup, else you can add a static IP using a `systemd-network .network` configuration file.

2.2.2 Basic system: SADM

We will now start to install and configure everything that is Prologin-specific. The bootstrap script has already copied the sadm repository to `/root/sadm`. We will now use a script that installs the dependencies that have to be present on all system using sadm. We are running the script on `gw.prolo` and it will executed on every systems: `rhfs`, `web`, `rfs`.

```
cd /root/sadm/install_scripts
./setup_sadm.sh
```

This script also creates a python virtual environment. Each time you log into a new system, activate the virtualenv:

```
source /opt/prologin/venv/bin/activate
```

2.2.3 Basic system: gw

Once the system is SADM-ready, perform installs specific to `gw.prolo`:

```
./setup_gw.sh
```

2.2.4 Gateway network configuration

`gw` has multiple static IPs used in our local network:

- 192.168.1.254/23 used to communicate with both the services and the users
- 192.168.250.254/24 used to communicate with aliens (aka. machines not in mdb)

It also has IP to communicate with the outside world:

- 10.?.?.?/8 static IP given by the bocal to communicate with the bocal gateway
- 163.5.??.??/16 WAN IP given by the CRI

The network interface(s) are configured using `systemd-networkd`. Our configuration files are stored in `etc/systemd/network/` and will be installed in `/etc/systemd/network` during the next step.

Two files must be modified to match the hardware of the machine:

- `etc/systemd/network/10-gw.link`: edit the `MACAddress` field of the file to set the MAC address of your NIC.
- `etc/systemd/network/10-gw.network`: we enable DHCP configuration and set the local network static IPs. You can edit this file to add more static IPs or set the gateway you want to use.

For this step, we use the following systemd services:

- From systemd: `systemd-networkd.service`: does the network configuration, interface renaming, IP setting, DHCP getting, gateway configuring, you get the idea. This service is enabled by the Arch Linux bootstrap script.
- From sadm: `nic-configuration@.service`: network interface configuration, this service should be enabled for each of the interface on the system.
- From sadm: `conntack.service`: does the necessary logging to comply with the fact that we are responsible for what the users are doing when using our gateway to the internet.

For more information, see the systemd-networkd documentation.

Then, install them:

```
python install.py systemd_networkd_gw nic_configuration conntrack
# you can now edit the configuration files as previously described
systemctl enable --now systemd-networkd conntrack
# `prologin` is the name of the interface to apply the configuration
systemctl enable --now nic-configuration@prologin
```

At this point you should reboot and test your network configuration:

- Your network interfaces should be up (`ip link show` shoud show `state UP` for all interfaces but `lo`).
- The IP addresses (`ip address show`) are correctly set to their respective interfaces.
- Default route (`ip route show`) should be the CRI's gateway.
- **DNS is not working until you setup ``mdbdns``, so keep on!**

2.2.5 Setup PostgreSQL on gw

First we need a database to store all kind of data we have to manipulate. There are two main PostgreSQL databases systems running the final, the first is on `gw` and the second is on `web`. The one on `gw` is used for sadm critical data such as the list of machines and users, while the one on `web` is used for contest related data.

By running this command, you will install the configuration files and start the database system:

```
cd sadm
python install.py postgresql
systemctl enable --now postgresql
```

To test this step:

```
$ systemctl status postgresql.service
● postgresql.service - PostgreSQL database server
   Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; vendor␣
↪preset: disabled)
```

(continues on next page)

(continued from previous page)

```
   Active: active (running) since Sun 2016-09-25 15:36:43 CEST; 2h 29min ago
 Main PID: 34 (postgres)
   CGroup: /machine.slice/machine-gw.scope/system.slice/postgresql.service
           ├─34 /usr/bin/postgres -D /var/lib/postgres/data
           ├─36 postgres: checkpointer process
           ├─37 postgres: writer process
           ├─38 postgres: wal writer process
           ├─39 postgres: autovacuum launcher process
           └─40 postgres: stats collector process
$ ss -nltp | grep postgres
LISTEN     0      128          *:5432                     *:*                   ␣
↪users:(("postgres",pid=34,fd=3))
LISTEN     0      128         :::5432                    :::*                   ␣
↪users:(("postgres",pid=34,fd=4))
$ su - postgres -c 'psql -c \\l'
                                  List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
(3 rows)
```

2.2.6 mdb

We now have a basic environment to start setting up services on our gateway server. We're going to start by installing `mdb` and configuring `nginx` as a reverse proxy for this application.

First, we need to install `nginx`:

```
$ pacman -S nginx
```

In order to test if `mdb` is working properly, we need to go to query `http://mdb/` with a command line tool like `curl`. However, to get DNS working, we need `mdbdns`, which needs `mdbsync`, which needs `mdb`. As a temporary workaround, we're going to add `mdb` to our `/etc/hosts` file:

```
echo '127.0.0.1 mdb' >> /etc/hosts
```

Similarly, our nginx configuration depends on the `sso` host to resolve, without what nginx will refuse to start. We have to define it statically too:

```
echo '127.0.0.1 sso' >> /etc/hosts
```

Then install `mdb`. Fortunately, a very simple script is provided with the application in order to setup what it requires:

```
# You can then proceed to install
python install.py mdb
mv /etc/nginx/nginx.conf{.new,}
# ^ To replace the default configuration by our own.
```

Note: You don't have to create super users for `mdb` or `udb` using the `manage.py` command. The root users you will add to `udb` will be super user and replicated to `mdb`. If you want to modify the databases before that, use `manage.py`

shell.

This command installed the mdb application to /var/prologin/mdb and installed the systemd and nginx configuration files required to run the application.

You should be able to start mdb and nginx like this:

```
systemctl enable --now mdb
systemctl enable --now nginx
```

Now you should get an empty list when querying /call/query:

```
curl http://mdb/call/query
# Should return []
```

Congratulations, mdb is installed and working properly!

You can check the journal for nginx, and should see:

```
journalctl -fu nginx
...
Mar 22 20:12:12 gw systemd[1]: Started Openresty, a powerful web app server,␣
↪extending nginx with lua scripting.
Mar 22 20:14:13 gw nginx[46]: 2017/03/22 20:14:13 [error] 137#0: *1 connect() failed␣
↪(111: Connection refused), client: 127.0.0.1, server: mdb, request: "GET /query␣
↪HTTP/1.1", host: "mdb"
Mar 22 20:14:13 gw nginx[46]: 2017/03/22 20:14:13 [error] 137#0: *1 [lua] access.
↪lua:77: SSO: could not query presenced: failed to join remote: connection refused,␣
↪client: 127.0.0.1, server: mdb, request: "GET /query HTTP/1.1", host: "mdb"
```

Note: nginx will log an error (connect() failed (111: Connection refused), client: 127.0.0.1, server: mdb) when attempting to connect to the upstream, this is normal and should only happen for the first time you connect to a service.

2.2.7 mdbsync

The next step now is to setup mdbsync. mdbsync is a Tornado web server used for applications that need to react on mdb updates. The DHCP and DNS config generation scripts use it to automatically update the configuration when mdb changes. Once again, setting up mdbsync is pretty easy:

```
python install.py mdbsync
systemctl enable --now mdbsync
systemctl reload nginx
echo '127.0.0.1 mdbsync' >> /etc/hosts
```

To check if mdbsync is working, try to register for updates:

```
python -c 'import prologin.mdbsync.client; prologin.mdbsync.client.connect().poll_
↪updates(print)'
# Should print {} {} and wait for updates
```

2.2.8 mdbdns

`mdbdns` gets updates from `mdbsync` and regenerates the DNS configuration. Once again, an installation script is provided:

```
python install.py mdbdns
mv /etc/named.conf{.new,}
# ^ To replace the default configuration by our own.
systemctl enable --now mdbdns
systemctl enable --now named
```

We now need to add a record in `mdb` for our current machine, `gw`, so that DNS configuration can be generated:

```
cd /var/prologin/mdb
python manage.py addmachine --hostname gw --mac 11:22:33:44:55:66 \
    --ip 192.168.1.254 --rfs 0 --hfs 0 --mtype service --room pasteur \
    --aliases mdb,mdbsync,ns,netboot,udb,udbsync,presencesync,ntp,sso
```

Once this is done, `mdbdns` should have automagically regenerated the DNS configuration:

```
host mdb.prolo 127.0.0.1
# Should return 192.168.1.254
```

You can now remove the lines related to `mdb`, `mdbsync` and `sso` from your `/etc/hosts` file.

2.2.9 mdbdhcp

`mdbdhcp` works just like `mdbdns`, but for DHCP. You must edit `dhcpd.conf` to add an empty subnet for the IP given by the Bocal. If it is on the same interface as 192.168.0.0/23, add it inside the `shared-network` `prolo-lan`, else add it to a new `shared-network`:

```
python install.py mdbdhcp
mv /etc/dhcpd.conf{.new,}
# ^ To replace the default configuration by our own.
$EDITOR /etc/dhcpd.conf
systemctl enable --now mdbdhcp
```

The DHCP server will provide the Arch Linux install medium for all the servers, for that, download the Netboot Live System:

```
# See https://www.archlinux.org/releng/netboot/
wget https://www.archlinux.org/static/netboot/ipxe.pxe -O /srv/tftp/arch.kpxe
```

Start the DHCP server:

```
systemctl enable --now dhcpd4
```

Note: `gw` needs to have `192.168.1.254/23` as a static IP or else `dhcpd` will not start.

To test this step:

```
$ systemctl status dhcpd4
● dhcpd4.service - IPv4 DHCP server
   Loaded: loaded (/usr/lib/systemd/system/dhcpd4.service; enabled; vendor preset:␣
↪disabled)
```

(continues on next page)

(continued from previous page)

```
   Active: active (running) since Sun 2016-09-25 18:41:57 CEST; 6s ago
  Process: 1552 ExecStart=/usr/bin/dhcpd -4 -q -cf /etc/dhcpd.conf -pf /run/dhcpd4.
↪pid (code=exited, status=0/SUCCESS)
 Main PID: 1553 (dhcpd)
   CGroup: /machine.slice/machine-gw.scope/system.slice/dhcpd4.service
           └─1553 /usr/bin/dhcpd -4 -q -cf /etc/dhcpd.conf -pf /run/dhcpd4.pid

Sep 25 18:41:57 gw systemd[1]: Starting IPv4 DHCP server...
Sep 25 18:41:57 gw dhcpd[1552]: Source compiled to use binary-leases
Sep 25 18:41:57 gw dhcpd[1552]: Wrote 0 deleted host decls to leases file.
Sep 25 18:41:57 gw dhcpd[1552]: Wrote 0 new dynamic host decls to leases file.
Sep 25 18:41:57 gw dhcpd[1552]: Wrote 0 leases to leases file.
Sep 25 18:41:57 gw dhcpd[1553]: Server starting service.
Sep 25 18:41:57 gw systemd[1]: Started IPv4 DHCP server.
$ ss -a -p | grep dhcpd
p_raw  UNCONN     0      0       *:host0                  *                      ␣
↪users:(("dhcpd",pid=1553,fd=5))
u_dgr  UNCONN     0      0       * 7838541              * 7790415              ␣
↪users:(("dhcpd",pid=1553,fd=3))
raw    UNCONN     0      0       *:icmp                   *:*                    ␣
↪users:(("dhcpd",pid=1553,fd=4))
udp    UNCONN     0      0       *:64977                  *:*                    ␣
↪users:(("dhcpd",pid=1553,fd=20))
udp    UNCONN     0      0       *:bootps                 *:*                    ␣
↪users:(("dhcpd",pid=1553,fd=7))
udp    UNCONN     0      0      :::57562                 :::*                    ␣
↪users:(("dhcpd",pid=1553,fd=21))
```

2.2.10 netboot

Netboot is a small HTTP service used to handle interactions with the PXE boot script: machine registration and serving kernel files. Once again, very simple setup:

```
python install.py netboot
systemctl enable --now netboot
systemctl reload nginx
```

2.2.11 TFTP

The TFTP server is used by the PXE clients to fetch the first stage of the boot chain: the iPXE binary (more on that in the next section). We simply setup `tftp-hpa`:

```
systemctl enable --now tftpd.socket
```

The TFTP server will serve files from `/srv/tftp`. We'll put files in this directory in the next step, and then during the setup of the exported NFS system.

2.2.12 iPXE bootrom

The iPXE bootrom is an integral part of the boot chain for user machines. It is loaded by the machine BIOS via PXE and is responsible for booting the Linux kernel using the nearest RFS. It also handles registering the machine in the MDB if needed.

We need a special version of iPXE supporting the LLDP protocol to speed up machine registration. We have a pre-built version of the PXE image in our Arch Linux repository:

```
pacman -S ipxe-sadm-git
```

This package installs the PXE image as `/srv/tftp/prologin.kpxe`.

2.2.13 udb

Install `udb` using the `install.py` recipe:

```
python install.py udb
```

Enable the service:

```
systemctl enable --now udb
systemctl reload nginx
```

You can then import all contestants information to `udb` using the `batchimport` command:

```
cd /var/prologin/udb
python manage.py batchimport --file=/root/finalistes.txt
```

The password sheet data can then be generated with this command, then printed by someone else:

```
python manage.py pwdsheetdata --type=user > /root/user_pwdsheet_data
```

Then do the same for organizers:

```
python manage.py batchimport --logins --type=orga --pwdlen=10 \
    --file=/root/orgas.txt
python manage.py pwdsheetdata --type=orga > /root/orga_pwdsheet_data
```

Then for roots:

```
python manage.py batchimport --logins --type=root --pwdlen=10 \
    --file=/root/roots.txt
python manage.py pwdsheetdata --type=root > /root/root_pwdsheet_data
```

2.2.14 udbsync

`usbsync` is a server that pushes updates of the user list.

Again, use the `install.py` recipe:

```
python install.py udbsync

systemctl enable --now udbsync
systemctl reload nginx
```

We can then configure udbsync clients:

```
python install.py udbsync_django udbsync_rootssh
systemctl enable --now udbsync_django@mdb
systemctl enable --now udbsync_django@udb
systemctl enable --now udbsync_rootssh
```

Note: Adding all the users to the sqlite databases is slow will lock them. You will have to wait a bit for mdb and udb to sync their user databases.

2.2.15 presencesync

presencesync manages the list of logged users. It authorizes user logins and maintain the list of logged users using pings from the presenced daemon running in the NFS exported systems.

Once again:

```
python install.py presencesync

systemctl enable --now presencesync
systemctl reload nginx
```

2.2.16 presencesync_sso

This listens to both presencesync and mdb updates and maintains a double mapping ip addr → machine hostname → logged-in username. This provides a way of knowing which user is logged on what machine by its IP address. This is used by nginx SSO to translate request IPs to logged-in username.

We expose an HTTP endpoint on gw nginx at http://sso/. Install the daemon and nginx config with:

```
python install.py presencesync_sso

systemctl enable --now presencesync_sso
systemctl reload nginx
```

All services that support SSO should already have the proper stubs in their respective nginx config. See the comments in etc/nginx/sso/{handler,protect} for how to use these stubs in new HTTP endpoints.

Debugging SSO

Typical symptoms of an incorrect SSO setup are:

- you're not automatically logged-in on SSO-enabled websites such as http://udb or http://concours
- nginx logs show entries mentioning __sso_auth or something about not being able to connect to some sso upstream

Your best chance at debugging this is to check the reply headers in your browser inspection tool.

- if there is not any of the headers described below, it means your service is not SSO-enabled, ie. doesn't contain the stubs mentioned above. Fix that.
- X-SSO-Backend-Status should be working, otherwise it means nginx cannot reach the SSO endpoint; in that case check that presencesync_sso works and http://sso is reachable.

- `X-SSO-Status` should be `authenticated` and `X-SSO-User` should be filled-in; if the website is not in a logged-in state, it means SSO is working but the website does not understand, or doesn't correctly handle the SSO headers. Maybe it is configured to get the user from a different header eg. `Remote-User`? Fix the website.
- if `X-SSO-Status` is `missing header`, it means nginx is not sending the real IP address making the request; are you missing `include sso/handler`?
- if `X-SSO-Status` is `unknown IP`, it means `presencesync_sso` couldn't resolve the machine hostname from its IP; check the IP exists in http://mdb and that `presencesync_sso` is receiving `mdb` updates.
- if `X-SSO-Status` is `logged-out machine`, it means `presencesync_sso` believes no one is logged-in the machine from which you do the requests; check that `presencesync` knows about the session (eg. using http://map/) and that `presencesync_sso` is receiving `presencesync` updates.

2.2.17 iptables

Note: If the upstream of `gw` is on a separate NIC you should replace `etc/iptables.save` with `etc/iptables_upstream_nic.save`

The name of the interface is hardcoded in the iptables configuration, you must edit it to match your setup:

```
$EDITOR etc/iptables.save
```

Setup the iptables rules and ipset creation for users allowed internet acces:

```
python install.py firewall
systemctl enable --now firewall
```

And the service that updates these rules:

```
python install.py presencesync_firewall
systemctl enable --now presencesync_firewall
```

2.3 Step 2: file storage

rhfs naming scheme

A rhfs has two NICs and is connected to two switches, there is therefore two `hfs-server` running on one rhfs machine, each with a different id. The hostname of the rhfs that hosts hfs `0` and hfs `1` will be: `rhfs01`.

A RHFS, for "root/home file server", has the following specifications:

- It is connected to two switches, handling two separates L2 segments. As such, the machine on a L2 segment is only 1 switch away from it RHFS. This is a good thing as it reduces the network latency, reduces the risk if one the switches in the room fails and simplyfies debugging network issues. It also mean that a RHFS will be physically near the machines it handles, pretty useful for debugging, although you will mostly work using SSH.
- Two NICs configured using DHCP, each of them connected to a different switch.
- Two disks in RAID1 setup, same as gw.

To bootstrap a rhfs, `rhfs01` for example, follow this procedure:

1. Boot the machine using PXE and register it into `mdb` as `rhfs01`.
2. Go to `mdb/` and add aliases for the NIC you just registered: `rhfs,rhfs0,hfs0,rfs0`. Also add another machine : `rhfs1` with the MAC address of the second NIC in the rhfs, it shoud have the following aliases: `hfs1,rfs1`.
3. Reboot the machine and boot an Arch Linux install medium.
4. Follow the same first setup step as for `gw`: see *Basic system: file system setup*.

2.3.1 Registering the switches

To be able to register the machines easily, we can register all the switches in MDB. By using the LLDP protocol, when registering the machines, they will be able to see which switch they are linked to and automatically guess the matching RHFS server.

On each rhfs, run the following command:

```
networkctl lldp
```

You should see an LLDP table like this:

```
LINK    CHASSIS ID          SYSTEM NAME   CAPS        PORT ID           PORT␣
↪DESCRIPTION
rhfs0   68:b5:99:9f:45:40   sw-kb-past-2  ..b........ 12                12
rhfs1   c0:91:34:c3:02:00   sw-kb-pas-3   ..b........ 22                22
```

This means the "rhfs0" interface of rhfs01 is linked to a switch named sw-kb-past-2 with a Chassis ID of 68:b5:99:9f:45:40.

After running this on all the rhfs, you should be able to establish a mapping like this:

```
rhfs0 -> sw-kb-past-2 (68:b5:99:9f:45:40)
rhfs1 -> sw-kb-pas-3 (c0:91:34:c3:02:00)
rhfs2 -> sw-kb-pas-4 (00:16:b9:c5:25:60)
rhfs3 -> sw-pas-5 (00:16:b9:c5:84:e0)
rhfs4 -> sw-kb-pas-6 (00:14:38:67:f7:e0)
rhfs5 -> sw-kb-pas-7 (00:1b:3f:5b:8c:a0)
```

You can register all those switches [in MDB](http://mdb/mdb/switch/). Click on "add switch", with the name of the switch like `sw-kb-past-2`, the chassis ID like `68:b5:99:9f:45:40`, and put the number of the interface in the RFS and HFS field (i.e if it's on the interface `rhfs0`, put 0 in both fields).

2.4 Step 3: booting the user machines

Note: if you are good at typing on two keyboards at once, or you have a spare root doing nothing, this step can be done in parallel with step 4.

2.4.1 Installing the RHFS

The basic install process is already documented through the ArchLinux Diskless Installation. For conveniance, use:

```
# Setup the rhfs server, install the exported rootfs
( cd ./install_scripts; ./setup_rfs.sh )
# Setup the exported rootfs
python install.py rfs_nfs_archlinux
```

Configure the exported rootfs for SADM and network booting. This scripts will chroot into the exported file system and run the `setup_sadm.sh` script.

```
python install.py rfs_nfs_sadm
```

The installation script will bootstrap a basic Arch Linux system in `/export/nfsroot_staging` using the common Arch Linux install script you already used for bootstraping `gw` and `rhfs`. It also adds a prologin hook that creates tmpfs at `/var/{log,tmp,spool/mail}`, installs libprologin and enables some sadm services.

We can now finish the basic RFS setup and export the NFS:

```
python install.py rfs
# Enable the services we just installed:
for svc in {udbsync_passwd{,_nfsroot},udbsync_rootssh,rpcbind,nfs-server}.service␣
↪rootssh.path; do
  echo "[-] Enable $svc"
  systemctl enable --now "$svc"
done
```

Once done, we need to copy the the kernel and initramfs from `rhfs` to `gw`, where they will be fetched by the machines during PXE. We also need to copy nfsroot_staging to the `rfs{0,2,4,6}:/export/nfsroot_ro`.

To do so, run on rhfs01:

```
rfs/commit_staging.sh rhfs01 rhfs23 rhfs45 rhfs67
```

At this point the machines should boot and drop you to a login shell. We can now start to install a basic graphical session, with nice fonts and graphics:

```
python install.py rfs_nfs_packages_base
```

You can reboot a machine and it should display a graphical login manager. You still need to install the `hfs` to login as a user.

If you want a full RFS install, with all the code editors you can think of and awesome games, install the extra package list:

```
python install.py rfs_nfs_packages_extra
```

To install a new package:

```
pacman --root /export/nfsroot_staging -Sy package
# deploy the newly created root to rhfs{0,2,4,6}:/export/nfsroot_ro
/root/sadm/rsync_rfs.sh rfs0 rfs2 rfs4 rfs6
```

Note: *Never* use arch-chroot or systemd-nspawn on a live NFS export. This will bind the runtime server directories, which will be picked up by the NFS clients resulting in great and glorious system failures.

TODO: How to sync, hook to generate /var…

2.4.2 Setting up hfs

On `gw`, install the hfs database:

```
python install.py hfsdb
```

2.4.3 Start the hfs

On every `rhfs` machine, install the hfs server:

```
python install.py hfs
# Change HFS_ID to what you need
systemctl enable --now hfs@HFS_ID
```

Then, setup the skeleton of a user home:

```
cp -r STECHEC_BUILD_DIR/home_env /export/skeleton
```

Test procedure:

1. Boot a user machine
2. Log using a test account (create one if needed), a hfs should be created with the skeleton in it.
3. The desktop launches, the user can edit files and start programs
4. Close the session
5. Boot a user machine using an other hfs
6. Log using the same test account, the hfs should be be migrated.
7. The same desktop launches with modifications.

2.4.4 Forwarding of authorized_keys

On a rhfs, the service `udbsync_rootssh` (aka. `udbsync_clients.rootssh`) writes the ssh public keys of roots to `/root/.ssh/authorized_keys`. The unit `rootssh.path` watches this file, and on change starts the service `rootssh-copy` that updates the `authorized_keys` in the `/exports/nfsroot_ro`.

2.5 Step 4: Concours

2.5.1 Setup web

The web services will usually be set up on a separate machine from the `gw`, for availability and performance reasons (all services on `gw` are critical, so you wouldn't want to mount a NFS on it for example). This machine is named `web.prolo`.

Once again, register a server on mdb and set up a standard Arch system. Add the following alliases in `mdb`:

```
db,concours,wiki,bugs,redmine,docs,home,paste,map,masternode
```

You will want to ssh at this machine, so enable `udbync_rootssh`:

```
python install.py udbsync_rootssh
systemctl enable --now udbsync_rootssh
```

Then install another nginx instance:

```
pacman -S nginx
```

Then, install the nginx configuration from the repository:

```
python install.py nginxcfg
mv /etc/nginx/nginx.conf{.new,}
systemctl enable --now nginx
```

2.5.2 Setup PostgreSQL on web

Install and enable PostgreSQL:

```
python install.py postgresql
systemctl enable --now postgresql
```

2.5.3 concours

Note: Concours is a *contest* service. It won't be enabled by default. See *Enable contest services*.

Run the following commands:

```
python install.py concours
systemctl enable --now concours
systemctl enable --now udbsync_django@concours
systemctl reload nginx
```

You can verify that concours is working by visiting http://concours

2.6 Step 5: Setting up masternode and workernode

On `masternode` (usually, `web`):

```
python install.py masternode
systemctl enable --now masternode
```

`workernode` must be running on all the users machine, to do that we install it in the NFS export. The required packages are `stechec` and `stechec2-makefiles`. We will intall them using the `prologin` Arch Linux repository:

```
pacman -S prologin/stechec2 prologin/stechec2-makefiles -r /export/nfsroot_staging
```

Note: The rfs setup script (`setup_nfs_export.sh`, ran by `install.py rfs_nfs_sadm`) already ran the following commands, we still list them for reference.

Then, still for the users machines, install `workernode`:

```
systemd-nspawn -D /export/nfsroot_staging/
cd sadm
python install.py workernode
systemctl enable workernode
exit # get out of the chroot
```

You may now reboot a user machine and check that the service is started (`systemctl status workernode.service`) and that the worker is registered to the master.

You should now be able to upload matches to `concours/` (you have to enable it see , see *Enable contest services*), see them dispatched by `masternode` to `workernode` s and get the result.

2.7 Step 6: Switching to contest mode

Contest mode is the set of switches to block internet access to the users and give them access to the contest ressources.

2.7.1 Block internet access

Edit `/etc/prologin/presencesync_firewall.yml` and remove the `user` group, the restart `presencesync_firewall`.

2.7.2 Enable contest services

By default, most of the web services are hidden from the contestants. In order to show them, you must activate the "contest mode" in some service.

Edit `/etc/nginx/nginx.conf`, uncomment the following line:

```
# include services_contest/*.nginx;
```

2.8 Common tasks

2.8.1 Enable Single Sign-On

By default, SSO is disabled as it requires other dependencies to be up and running.

Edit `/etc/nginx/nginx.conf`, uncomment the following lines:

```
# lua_package_path '/etc/nginx/sso/?.lua;;';
# init_by_lua_file sso/init.lua;
# access_by_lua_file sso/access.lua;
```

2.8.2 Customize the wallpaper

To customize the desktop wallpaper, create a PNG file at the following location and *commit* the changes:

```
/export/nfsroot_staging/opt/prologin/wallpaper.png
```

The following DE are setup to use this file:

- i3
- awesome
- Plasma (aka. KDE)
- XFCE

Gnome-shell is still to be done.

2.8.3 Customize the SDDM logo

To customize the SDDM logo, replace the SVG file at the following location and *commit* the changes:

```
/export/nfsroot_staging/usr/share/sddm/themes/prologin/prologin-logo.svg
```

CHAPTER

THREE

MISC SERVICES

3.1 /sgoinfre

Setup a `rw` nfs export on a `misc` machine, performance or reliabiliy is not a priority for this service.

Install `nfs-utils` on *misc*.

Add the following line to /etc/exports:

```
/sgoinfre *(rw,insecure,squash_all,no_subtree_check,nohide)
```

Run the following commands on *misc*:

```
exportfs -arv
systemctl enable --now nfs-server
```

The following `systemd` service can be installed on the rhfs (in nfsroot):

```
# /etc/systemd/system/sgoinfre.mount
[Unit]
After=network-online.target

[Mount]
What=sgoinfre:/sgoinfre
Where=/sgoinfre
Options=nfsvers=4,nolock,noatime

[Install]
WantedBy=multi-user.target
```

Then enable the unit:

```
# systemctl enable --now sgoinfre.mount
```

3.2 doc

Setup:

```
python install.py docs
systemctl reload nginx
```

You have to retrieve the documentations of each language:

```
pacman -S wget unzip
su webservices - # So that the files have the right owner
cd /var/prologin/docs/languages
./get_docs.sh
```

You can now test the docs:

- Open `http://docs/languages/`
- Click on each language, you should see their documentation.

3.3 paste

We will setup dpaste: https://github.com/bartTC/dpaste:

```
pip install dpaste
python install.py paste

systemctl enable paste && systemctl start paste
systemctl reload nginx
```

3.4 Redmine

First, export some useful variables. Change them if needed:

```
export PHOME=/var/prologin
export PGHOST=web  # postgres host
export RUBYV=2.2.1
export RAILS_ENV=production
export REDMINE_LANG=fr
read -esp "Enter redmine db password (no ' please): " RMPSWD
```

Download and extract Redmine:

```
cd /tmp
wget http://www.redmine.org/releases/redmine-3.0.1.tar.gz
tar -xvz -C $PHOME -f redmine*.tar.gz
mv $PHOME/{redmine*,redmine}
```

Using RVM, let's install dependencies:

```
# Trust RVM keys
curl -sSL https://rvm.io/mpapis.asc | gpg2 --import -
curl -sSL https://get.rvm.io | bash -s stable
```

(continues on next page)

(continued from previous page)

```
source /etc/profile.d/rvm.sh
echo "gem: --no-document" >>$HOME/.gemrc
rvm install $RUBYV  # can be rather long
rvm alias create redmine $RUBYV
gem install bundler unicorn
```

Create the Redmine user and database:

```
sed -e s/DEFAULT_PASSWORD/$RMPSWD/ /root/sadm/sql/redmine.sql | su - postgres -c psql
```

Configure the Redmine database:

```
cat >$PHOME/redmine/config/database.yml <<EOF
# prologin redmine database
production:
  adapter: postgresql
  database: redmine
  host: $PGHOST
  username: redmine
  password: $RMPSWD
  encoding: utf8
EOF
```

We can now install Redmine:

```
cd $PHOME/redmine
bundle install --without development test rmagick
```

Some fixtures (these commands require the above env vars):

```
bundle exec rake generate_secret_token
bundle exec rake db:migrate
bundle exec rake redmine:load_default_data
```

Create some dirs and fix permissions:

```
mkdir -p $PHOME/redmine/{tmp,tmp/pdf,public/plugin_assets}
chown -R redmine:http $PHOME/redmine
chmod -R o-rwx $PHOME/redmine
chmod -R 755 $PHOME/redmine/{files,log,tmp,public/plugin_assets}
```

Install the SSO plugin:

```
( cd $PHOME/redmine/plugins && git clone https://github.com/prologin/redmine-sso-auth.
↪git )
```

Now it's time to install Redmine system configuration files. Ensure you are within the prologin virtualenv (`source /opt/prologin/venv/bin/activate`), then:

```
cd /root/sadm
python install.py redmine udbsync_redmine
```

Register the new plugins (SSO, IRC hook):

```
( cd $PHOME/redmine && exec rake redmine:plugins:migrate )
# Should display:
```

(continues on next page)

(continued from previous page)

```
# Migrating issues_json_socket_send (Redmine issues to socket JSON serialized)...
# Migrating redmine_sso_auth (SSO authentication plugin)...
```

Enable and start the services:

```
systemctl enable redmine && systemctl start redmine
systemctl enable udbsync_redmine && systemctl start udbsync_redmine
systemctl reload nginx
```

You should be able to access the brand new Redmine. There are some important configuration settings to change:

- Login at http://redmine/login with `admin` / `admin`
- Change password at http://redmine/my/password
- In http://redmine/settings?tab=authentication - Enable enforced authentication. - Set minimum password length to 0. - Disable lost password feature, account deletion and registration.
- In http://redmine/settings/plugin/redmine_sso_auth - Enable SSO. - If not already done, set environment variable to `HTTP_X_SSO_USER`. - Set search method to username.
- Configure a new project at http://redmine/projects/new The `Identifiant` **has to be ``prologin``** in order to vhosts to work.
- As soon as `udbsync_redmine` has finished its first sync, you should find the three groups (user, orga, root) at http://redmine/groups so you can give them special priviledges: click one, click the "Projets" tab, assign your "prologin" project to one of the roles. For instance: user → ∅, orga → developer, root → {manager, developer}

3.5 Homepage

The homepage links to all our web services. It is a simple Django app that allows adding links easily. Setup it using `install.py`:

```
python install.py homepage
systemctl enable homepage && systemctl start homepage
systemctl enable udbsync_django@homepage && systemctl start udbsync_django@homepage
```

You can then add links to the homepage by going to http://homepage/admin.

3.6 DJ-Ango

See dj_ango README: https://bitbucket.org/Zeletochoy/dj-ango/

3.7 IRC

Install the ircd, then install the config:

```
pacman -S unrealircd
python install.py ircd
mv /etc/unrealircd/unrealircd.conf{.new,}
```

Change the OPER password in the config:

```
vim /etc/unrealircd/unrealircd.conf
```

Then enable and start the IRCd:

```
systemctl enable --now unrealircd
```

Now you need to enable the SOCKS tunnel so that IRC is available from the outside. First, generate a ssh key in misc, and add it to an user of the public-facing server (e.g prologin.org):

```
ssh-keygen -t ed25519 -q -N "" < /dev/zero
ssh-copy-id dev@prologin.org
```

Then, enable and start the IRC gatessh:

```
systemctl enable --now irc_gatessh
```

3.7.1 IRC issues bot

Once both IRC and Redmine are installed, you can also install the IRC bot that warns about new issues:

```
python install.py irc_redmine_issues
systemctl enable --now irc_redmine_issues
```

CHAPTER

FOUR

MONITORING

Monitoring is the art of knowning when something fails, and getting as much information as possible to solve the issue.

We use prometheus as our metrics monitoring backend and grafana for the dashboards. We use elasticsearch to store logs and kibana to search through them.

We will use a separate machine for monitoring as we want to isolate it from the core services, because we don't want the monitoring workload to impact other services, and vice versa. The system is installed with the same base Arch Linux configuration as the other servers.

4.1 Setup

To make a good monitoring system, mix the following ingredients, in that order:

1. `bootstrap_arch_linux.sh`
2. `setup_monitoring.sh`
3. `python install.py prometheus`
4. `systemctl enable --now prometheus`
5. `python install.py grafana`
6. `systemctl enable --now grafana`

4.2 Monitoring services

Most SADM services come with built-in monitoring and should be monitored as soon as prometheus is started.

The following endpoints are availables:

- `http://udb/metrics`
- `http://mdb/metrics`
- `http://concours/metrics`
- `http://masternode:9021`
- `http://presencesync:9030`
- hfs: each hfs exports its metrics on `http://hfsx:9030`
- workernode: each workernode exports its metrics on `http://MACHINE:9020`.

4.3 Grafana configuration

In a nutshell:

1. Install the `grafana` package.
2. Copy the SADM configuration file: `etc/grafana/grafana.ini`.
3. Enable and start the `grafana` service
4. Copy the nginx configuration: `etc/nginx/services/grafana.nginx`
5. Open http://grafana/, login and import the SADM dashboards from `etc/grafana`.

Todo: automate the process above

4.4 Monitoring screen how-to

Start multiple `chromium --app http://grafana/` to open a monitoring web view.

We look at both the `System` and `Masternode` dashboards from grafana.

Disable the screen saver and DPMS using on the monitoring display using:

```
$ xset -dpms
$ xset s off
```

4.5 Log monitoring

On monitoring:

```
$ pacman -S elasticsearch kibana
$ systemctl enable --now elasticsearch kibana
```

In the kibana web UI, go to the dev tools tab and run:

```
# Make sure the index isn't there
DELETE /logs

# Create the index
PUT /logs

PUT logs/_mapping
{
  "properties": {
    "REALTIME_TIMESTAMP": {
      "type": "date",
      "format": "epoch_millis"
    }
  }
}
```

It creates an index called logs, as well as proper metadata for time filtering.

Install https://github.com/multun/journal-upload-aggregator on the monitoring server, and *please do not* configure nginx as a front-end on `journal-aggregator`. Don't forget to add the alias in `mdb`.

On the machines that need to be monitored, create `/etc/systemd/journal-upload.conf`:

```
[Upload]
Url=http://journal-aggregator:20200/gateway
```

If still not fixed, also create `/etc/systemd/system/systemd-journal-upload.service.d/restart.conf`:

```
[Service]
Restart=on-failure
RestartSec=4
```

Then:

```
$ systemctl enable --now systemd-journal-upload
```

As an useful first request:

```
not SYSTEMD_USER_SLICE:* and (error or (PRIORITY < 5) or (EXIT_STATUS:* and not EXIT_
↪STATUS:0))
```

This request filters non-user errors.

CHAPTER

FIVE

ARCH LINUX REPOSITORY

Prologin has setup an Arch Linux package repository to ease of use of custom packages and AUR content.

5.1 Usage

Add the following section to the `/etc/pacman.conf` file:

```
[prologin]
Server = https://repo.prologin.org/
```

Then, trust the repository signing keys:

```
$ wget https://repo.prologin.org/prologin.pub
$ pacman-key --add prologin.pub
$ pacman-key --lsign-key prologin
```

Finally, test the repository:

```
$ pacman -Sy
```

You should see "prologin" in the list of synchronized package databases.

5.2 SADM related packages

Some packages are key parts of the SADM architecture. They should always be the lastest revision possible. The packages we maintain are in the `pkg` folder.

5.3 Uploading packages

Only the owner of the repository's private key and ssh access to repo@prologin.org can upload packages.

To import the private key to your keystore:

```
$ ssh repo@prologin.org 'gpg --export-secret-keys --armor␣
↪F4592F5F00D9EA8279AE25190312438E8809C743' | gpg --import
$ gpg --edit-key F4592F5F00D9EA8279AE25190312438E8809C743
```

Trust fully the key.

Then, build the package you want to upload locally using `makepkg`. Once the package is built, use `pkg/upload2repo.sh` to sign it, update the database and upload it.

Example usage:

```
$ cd quake3-pak0.pk3
$ makepkg
$ ~/rosa-sadm/pkg/upload2repo.sh quake3-pak0.pk3-1-1-x86_64.pkg.tar.xz
```

You can then install the package like any other:

```
# pacman -Sy quake3-pak0.pk3
$ quake3
```

Enjoy!

5.4 More information

The repository content is stored in `rosa:~repo/www`. Use your Prologin SADM credentials when asked for a password or a passphrase.

5.5 Troubleshooting

5.5.1 Invalid signature of a database or a package

This should not happen. If it does, find the broken signature and re-sign the file using `gpg --sign`. You must also investigate why an invalid signature was generated.

CHAPTER

SIX

COOKBOOK

All the things you might need to do as an organizer or a root are documented here.

6.1 Server setup

Here is a list of things to remember when setting up servers:

- Use ssh as soon as possible.
- Work in a tmux session, this allows any other root to take over your work if needed.
- Use only one user and one shell (bash) and setup an infinite history. This, http://stackoverflow.com/a/19533853 is already installed by the rfs scripts. Doing that will document what you and the other admins are doing during the contest.

6.2 Testing on qemu/libvirt

Here are some notes:

- Do not use the spice graphical console for setting up servers, use the serial line. For syslinux it is `serial 0` at the top of `syslinux.cfg` and for Linux `console=ttyS0` on the cmd line of the kernel in `syslinux.cfg`.
- For best performance use the VirtIO devices (disk, NIC), this should already be configured if you used `virt-install` to create the machine.
- For user machines, use the QXL driver for best performance with SPICE.

6.3 User related operations

Most of the operations are made very simple with the use of `udb`. If you are an organizer, you can access `udb` in read only mode. If you are a root, you obviously have write access too.

`udb` displays the information (including passwords) of every contestant to organizers. Organizers can't see the information of other organizers or roots.

All services should be using `udb` for authentication. Synchronization might take up to 5 minutes (usually only one minute) if anything is changed.

Giving back his password to a contestant First of all, make sure to ask the contestant for his badge, which he should always have on him. Use the name from the badge to look up the user in the `udb`. The password should be visible there.

Adding an organizer **Root only**. Go to `udb` and add a user with type `orga`.

Send an announce Connect to the IRC server, join the #notify channel, and send a message formatted like this:

```
!announce <expiration-delay> <message>
```

Example:

```
!announce 12 The lunch is ready!
```

Will create an announce which will stay for 12 minutes on the users's desktops. Note that the delay will default to 2 when not specified:

```
!announce No milk today :(
```

6.4 Machine registration

`mdb` contains the information of all machines on the contest LANs. If a machine is not in `mdb`, it is considered an alien and won't be able to access the network.

All of these operations are **root only**. Organizers can't access the `mdb` administration interface.

Adding a user machine to the network In the `mdb` configuration, authorize self registration by adding a VolatileSetting `allow_self_registration` to true. Netboot the user machine - it should ask for registration details. After the details have been entered, the machine should reboot to the user environment. Disable `allow_self_registration` when you're done.

Adding a machine we don't manage to the user network Some organizers may want to use their laptop. Ask them for their MAC address and the hostname they want. Finally, insert a `mdb` machine record with machine type `orga` using the IP address you manually allocated (if you set the last allocation to 100, you should assign the IP .100). Wait a minute for the DHCP configuration to be synced, and connect the laptop to the network.

6.5 Network FS related operations

Two kind of network file systems are used during the finals, the first one is the Root File System: RFS, the second is the Home File System: HFS. The current setup is that a server is both a RFS and a HFS node.

The RFS is a read-only NFS mounted as a rootnfs in Linux. It is replicated over multiple servers to ensure minimum latency over the network.

The HFS is a read-write, exclusive, user-specific export of their home. In other words, each user has it's own personal space that can only be mounted once at a time. The HFS exports are sharded over multiple servers.

6.5.1 Resetting the hfs

If you need to delete every `/home` created by the hfs, simply delete all nbd files in `/export/hfs/` on all HFS servers and delete entries in the `user_location` table of the hfs' database:

```
# For each hfs instance
rm /export/hfs/*.nbd

echo 'delete from user_location;' | su - postgres -c 'psql hfs'
```

6.5.2 Remove a RAID 1

The first step is to deactivate and remove the volume group:

```
vgchange -a n data
vgremove data
```

Then you have to actually deconstruct the RAID array and zero the superblock of each device:

```
mdadm --stop /dev/md0
mdadm --remove /dev/md0
mdadm --zero-superblock /dev/sda2
mdadm --zero-superblock /dev/sdb2
```

If you want to erase the remaining ext4 filesystem on thoses devices, you can use fdisk.

CHAPTER

SEVEN

RUNNING THE WEBSITES WITHOUT A COMPLETE SADM SETUP

When hacking on the various web services included in SADM, it is not necessary to setup a full-blown SADM infrastructure. Typically, when making the design of `concours` website for a new edition, only a small Django setup has to be completed.

1. Clone SADM and cd into it:

```
git clone https://github.com/prologin/sadm
cd sadm
```

2. Configure the website:

```
# for the 'concours' site
vim etc/prologin/concours.yml
```

   Refer below for a guide of values to adapt depending on the website being worked on.

3. Create a venv (don't activate it yet):

```
python -m venv .venv
```

4. Add the configuration path to `.venv/bin/activate` so it is automatically set up when you activate the venv:

```
echo "export CFG_DIR='$PWD/etc/prologin'" >> .venv/bin/activate
```

5. Activate the venv, install the requirements:

```
source .venv/bin/activate
pip install -r requirements.txt -e .
```

6. Apply the migrations, create a super-user and run:

```
# for the 'concours' site
cd django/concours
python manage.py migrate
python manage.py createsuperuser --username prologin --email x@prologin.org
# fill in the password;
python manage.py runserver
```

   Go to http://localhost:8000/, use `prologin` and the password you just chose to log in.

7.1 Working on concours

7.1.1 Configuration

Customize etc/prologin/concours.yml with the following:

db.default The easiest way is to use SQLite:

```
ENGINE: django.db.backends.sqlite3
NAME: concours.sqlite
```

contest.game Use the correct year, typically prologin2018 for the 2018 edition.

contest.directory Use a writable directory, eg. /tmp/prologin/concours_shared.

website.static_path Put the absolute path to prologin<year>/www/static or whatever directory is used for this edition.

Other contest entries (eg. use_maps) Adapt to the correct settings for this edition.

7.1.2 Importing a stechec2 dump for testing

When developing the Javascript replay and other features, you might need to import test dumps that can be loaded on the website.

While in the correct virtualenv:

```
cd django/concours
python manage.py import_dump /path/to/my/dump.json
```

This will create a dummy match with zero players and no map, that will successfully load on the dedicated URL. The match detail URL output by this command will only work in the default setup where manage.py runserver is used on localhost:8000. Adapt the host/port if needed.

CHAPTER

EIGHT

CONTAINER SETUP FOR SADM

This page explains how to run and test the Prologin SADM infrastruction using containers.

Note: TL;DR run `container_setup_host.sh` then `container_setup_gw.sh` from `install_scripts/containers/`

8.1 Why containers?

Containers are lightweight isolation mechanisms. You can think of them as "starting a new userland on the same kernel", contrarily to virtual machines, where you "start a whole new system". You may know them from tools such as docker, kubernetes or rkt. In this guide we will use system-nspawn(1), which you may already have installed if you are using systemd. Its main advantages compared to other container managers are:

- Its simplicity. It does one thing well: configuring namespaces, the core of containers. No configuration file, daemon (other than systemd), managed filesystem or hidden network configuration. Everything is on the command line and all the options are in the man page.
- Its integrated with the systemd ecosystem. A container started with `systemd-nspawn` is registered and managable with *machinectl(1) <https://www.freedesktop.org/software/systemd/man/machinectl.html>*. You can use the `-M` of many systemd utilities (e.g. *systemctl*, *journalctl*) to control it.
- Its feature set. You can configure the filesystem mapping, network interfaces, resources limits and security properties you want. Just look at the man page to see the options.

Containers compare favorably to virtual machines on the following points:

- Startup speed. The containers share the devices of the host, these are already initialised and running therefore the boot time is reduced.
- Memory and CPU resources usage. No hypervisor and extra kernel overhead.
- Storage. The content of the container is stored in your existing file system and is actually completely editable from outside of the container. It's very useful for inspecting what's going on.
- Configuration. For *system-nspawn*, the only configuration you'll have is the command line.

8.2 Overview

This guides starts by discussing the virtual network setup, then we build and start the systems.

8.3 Networkd in your base system

The container setup requires systemd-networkd to be running on your system. That said, you might not want it to be managing your main network interfaces.

If you want to tell systemd-networkd to not manage your other interfaces, you can run this command:

```
cat >/etc/systemd/network/00-ignore-all.network <<EOF
[Match]
Name=!vz*

[Link]
Unmanaged=yes
EOF
```

8.4 Automated container setup

If you want to setup SADM in containers to test something else than the install procedure, you can use the automated container install scripts. They will create and manage the containers for you and perform a full SADM install as you would do manually. They are intended for automated and end-to-end tests of SADM.

Requirements:

- The host system should be Arch Linux. Experimental support has been added for non Arch Linux hosts (CoreOS) and will be used if the script detects you are not running Arch.
- For convenience, `/var/lib/machines` should be a btrfs volume. The scripts will run without that but you will not have the ability to restore intermediate snapshots of the install. Note that if you don't want to use a btrfs volume you can use:

  ```
  echo 'USE_BTRFS=false' > install_scripts/containers/container_setup.conf
  ```

To start, run the host setup script, you are strongly advised to check its content beforehand, as it does some substantial changes to your system setup:

```
cd install_scripts/containers/
./container_setup_host.sh
```

Then, run the container install scripts:

```
./container_setup_gw.sh
./container_setup_rhfs.sh
./container_setup_web.sh
./container_setup_pas-r11p11.sh
```

That's it!

You should be able to see the containers listed by `machinectl`, and you can get a shell on the system using `machinectl shell CONTAINER_NAME`.

8.5 What do the scripts do?

They automate setups of Arch Linux and SADM components in containers. The commands in the scripts are taken from the main setup documentation. We expect the container setup to follow the manual setup as strictly as possible.

8.6 BTRFS snapshots

Each stage of the system setups we are building can take a substantial amount of time to complete. To iterate faster we user file system snapshots at each stage so that the system can be rollback the stage just before what you want to test or debug.

Each `stage_*` shell function ends by a call to `container_snapshot $FUNCNAME`.

8.7 Cleaning up

If you want to clean up what these scripts did, you must stop the currently running containers. List the containers with `machinectl list` and `machinectl kill` all of them. You can then remove the containers' data by deleting the content of `/var/lib/machines`. List bind-mounted directories: `findmnt | grep /var/lib/machines/` and `umount` them. Then delete the BTRFS snapshots. List them using `btrfs subvolume list .` and delete them using `btrfs subvolume delete`.

8.8 Containers deep dive

As mentioned above, these scripts setup containers using `machinectl`. It's not necessary to understand how the containers work to test features in prologin-sadm, but you may encounter weird bugs caused by them. The following sections discuss some internals of the containers setup.

A key design decision is that the container setup should not require special cases added to the normal setup. This is to avoid bloating the code and keep it as simple as possible. The container setup can highlight potential fixes, for example how to make the setup more generic or how to decouple the services from the underlying system or network setup.

We note that containers do require special configuration. It should be applied in the container scripts themselves.

8.9 Virtual network setup

The first step consists in creating a virtual network. Doing it with containers is not that different compared to using virtual machines. We can still use bridge type interfaces to wire all the systems together, but we also have new possibilities, as the container is running on the same kernel as the host system.

One interesting thing is that we will be able to start one system as a container, let say gw.prolo, and others as virtual machines, for example the contestant systems, to test network boot for example.

We will use a bridge interface, the next problem to solve is to give this interface an uplink: a way to forward packets to the internet, and back again. To do that, we have multiple choices, here are two:

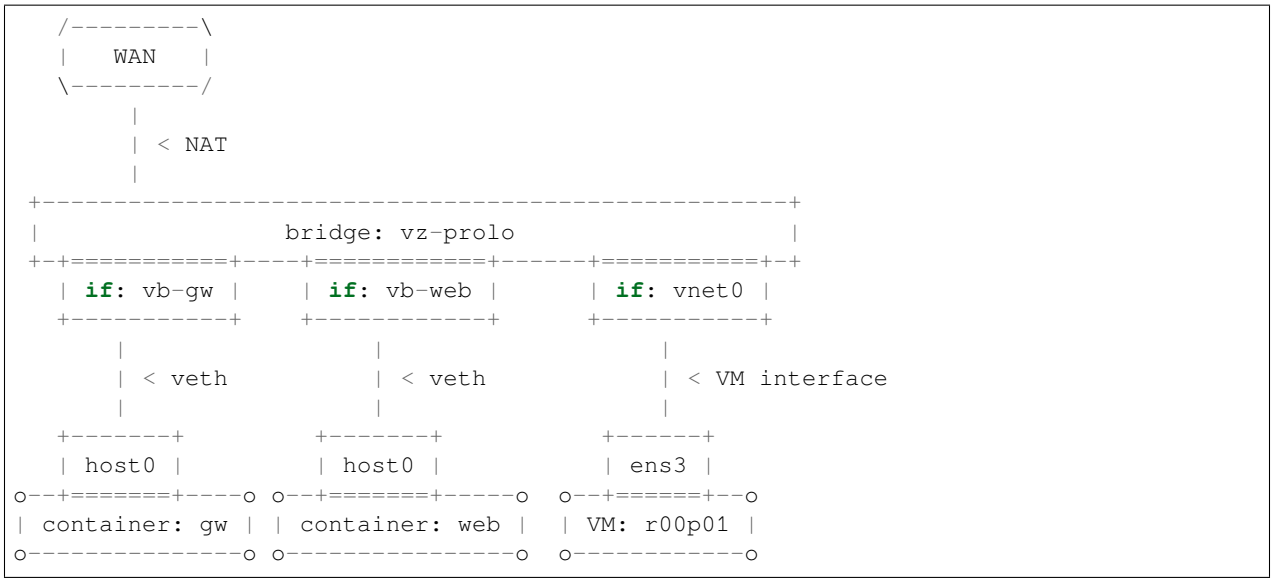
- Masquerade ("NAT") the `vz-prolo` bridge type interface behind your uplink. With this technique the packets arriving on `vz-prolo` will be rewritten, tracked and moved to your uplink to be routed as if they originated from it. The machines behind the NAT will not be accessible directly and you will have to setup port forwarding to access them from outside your system. From within your system they will be accessible directly using their local IP

address. In this guide we will use the "zone" network type of `systemd-nspawn` and `systemd-networkd` as the main system network manager. `systemd-networkd` will manage the NAT in iptables for us. Be careful, if you shadowed the `80-container-vz.network` rule with a catch-all (`Name=*`) `.network` configuration files, the NAT will not be created.

- Bridge your uplink interface with `vz-prolo`. This will have the bad effect to link your LAN, which is most likely already using your router DHCP server, to SADM network, which has its own DHCP server. Depending on various parameters your machine and those on your LAN might get IPs and DNS configuration from Prologin SADM. Be careful if you choose this option, as bridging your uplink will down the interface, `vz-prolo` will get an IP from your DHCP server if you use one and you may have to clean your routes to remove the old ones. It is still the fastest to setup, especially if you just want to give internet to a container. Note: as of 2016, some wireless drivers such as broadcom's `wl` do not support bridging 802.11 interfaces.

The NAT setup is simpler and more flexible, that's what we will use.

All the containers will be connected to their own L2 network using a bridge interface. This interface is managed by systemd, created when the first container using it is spawned. Here is a small diagram to explain how we want the network to look like:



Veth type interfaces what we will use) linked to a bridge will have the name `host0`. `systemd-networkd` provides a default configuration (`80-container-host0.network`) file that enable DHCP on them. With the NAT rule managed by `systemd-networkd` and that, the internet will be accessible out-of-the-box in the containers. The only remaining configuration to do being the DNS resolver (`/etc/resolv.conf`).

8.10 Setting up gw manually

Let's boot the first container: `gw`

Everything starts with an empty directory. This is where we will instantiate the file system used by `gw`:

```
$ mkdir gw
```

Use the Arch Linux install script from the sadm repository to populate it. Here is how to use it:

```
# ./install_scripts/bootstrap_arch_linux.sh /path/to/container machine_name ./file_
↪containing_plaintest_root_pass
```

We suggest storing the password in a text file. It's a good way to be able to to reproduce the setup quickly. If you don't want that, just create the file on the fly or delete it afterwards.

The first system we build is *gw*, so let's create the container accordingly. Run it as root:

```
# ./install_scripts/bootstrap_arch_linux.sh /path/to/gw gw ./plaintest_root_pass
```

Packages will get installed a few scripts run to configure the Arch Linux system. This is the same script we use for the bare metal or VM setup.

Then, start the container with a virtual ethernet interface connected to the `vz-prolo` network zone, a bridge interface managed by systemd, as well an ipvlan interface linked to your uplink:

```
# systemd-nspawn --boot --directory /path/to/gw --network-zone=prologin
```

Note: To exit the container, press 'ctrl+]' three time. `systemd-nspawn` told you that when it started, but there is good chance you missed it, so we are putting it here just for you :)

You should see systemd booting, all the units should be `OK` except `Create Volatile Files and Directories.` which fails because `/sys/` is mounted read-only by `systemd-nspawn`. After the startup you should get a login prompt. Login as *root* and check that you see the virtual interface named `host0` in the container using `ip link`:

```
# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: host0@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether e6:28:86:d2:de:6e brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

The host system should have two new interfaces:

- `vz-prolo`, a bridge type interface.
- `vb-gw`, a veth device whose master is `vz-prolo`, meaning it's wired in this bridge.

Both these interface have an extra `@...` suffix. It is not part of the interface name and simply indicates their peer interface.

If you are running `systemd-networkd` on your host system, with the default configuration files, the `vz-prolo` interface will get an IP from a private subnet and a `MASQUERADE` rule will be inserted into iptables. You can start `systemd-networkd` inside the container to get an IP in the `vz-prologin` network, which will be NAT'ed to your uplink.

For some reason `host0` cannot be renamed to `prologin` by a `systemd-networkd` .link file. What needs to be changed to account for that is:

- The firewall configuration

You can do the usual install, with the following changes:

- In `prologin.network`, in `[Match]`, set `Name=host0` to match the virtualized interface.

What will *not* work:

- Some services are disabled when run in a container, for example `systemd-timesyncd.service`.
- `nic-configuration@host0.service` will fail (`Cannot get device pause settings: Operation not supported`) as this is a virtual interface.

Note: When you exit the container everything you started inside it is killed. If you want a persistent container, run:

```
# systemd-run systemd-nspawn --keep-unit --boot --directory /full/path/to/gw --
↪network-zone=prologin
Running as unit run-r10cb0f7202be483b88ea75f6d3686ff6.service.
```

And then monitor it using the transient unit name:

```
# systemctl status run-r10cb0f7202be483b88ea75f6d3686ff6.service
```

8.11 Manual network configuration

This section is a do-it-yourself version of the `--network-veth --network-bridge=prologin` nspawn's arguments. The main advantage of doing so is that the interfaces are not deleted when the container is shut down. Its useful if you have iptables rules you want to keep.

First let's make sure we have ip forwarding enabled, without that the bridge will move packets around:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

We will create a bridge interface named `prologin` that will represent the isolated L2 network for SADM:

```
# ip link add prologin type bridge
```

You can now see the prologin interface using:

```
# ip link show
...
4: prologin: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT group default qlen 1000
```

For each system we want to start, we create a veth and plug one end to the bridge. For example for the `gw`:

```
# ip link add gw.local type veth peer name gw.bridge
# ip link show label 'gw*'
```

Here we create the two virtual ethernet interfaces, `gw.local@gw.local` and `gw.bridge@@gw.bridge`. On veth pairs, a packet arriving to one these interface is dispatched to the other. When manipulating them only the part of the name before the `@` is required, the other is just a reminder of what interface is at the other end.

Let's wire `gw.bridge` to the bridge:

```
# ip link set gw.bridge master prologin
```

You can see that the interface is connected to the bridge with the `master prologin` keyword on the following command:

```
$ ip link show gw.bridge
```

The interface is not running (`state DOWN`), we have to enable it:

```
# ip link set dev prologin up
```

CHAPTER

NINE

GOING FURTHER/DISCUSSION

What could make your container usage better?

- Use the `--overlay` option from `systemd-nspawn`. Have only one base Arch Linux distro and build other systems form it. It reduces the time to install and disk usage (if that's your concern).

CHAPTER

TEN

DISASTER RECOVERY

What to do when something bad and unexpected happen.

Here are the rules:

1. Diagnose root cause, don't fix the consequences of a bigger failure.
2. Balance the "quick" and the "dirty" of your fixes.
3. Always document clearly and precisely what was wrong and what you did.
4. Don't panic!

10.1 Disk failure

10.1.1 Hard fail

The `md` array will go into degraded mode. See `/proc/mdstat`.

If the disk breaks when the system is powered off, the `md` array will start in an inactive state and your will be dropped in the emergency shell. You will have to re-activate the array to continue booting:

```
$ mdadm --stop /dev/md127
$ mdadm --assemble
$ mount /dev/disk/by-label/YOUR_DISK_ROOT_LABEL new_root/
$ exit  # exit the emergency shell and continue the booting sequence
```

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search