systemd for Administrators, Part IX

Here's the ninth installment of my ongoing series on systemd for Administrators:

On /etc/sysconfig and /etc/default

So, here's a bit of an opinion piece on the `/etc/sysconfig/` and `/etc/default` directories that exist on the various distributions in one form or another, and why I believe their use should be faded out. Like everything I say on this blog what follows is just my personal opinion, and not the gospel and has nothing to do with the position of the Fedora project or my employer. The topic of `/etc/sysconfig` has been coming up in discussions over and over again. I hope with this blog story I can explain a bit what we as systemd upstream think about these files.

A few lines about the historical context: I wasn't around when /etc/sysconfig was introduced -- suffice to say it has been around on Red Hat and SUSE distributions since a long long time. Eventually /etc/default was introduced on Debian with very similar semantics. Many other distributions know a directory with similar semantics too, most of them call it either one or the other way. In fact, even other Unix-OSes sported a directory like this. (Such as SCO. If you are interested in the details, I am sure a Unix greybeard of your trust can fill in what I am leaving vague here.) So, even though a directory like this has been known widely on Linuxes and Unixes, it never has been standardized, neither in POSIX nor in LSB/FHS. These directories very much are something where distributions distuingish themselves from each other.

The semantics of `/etc/default` and `/etc/sysconfig` are very losely defined only. What almost all files stored in these directories have in common though is that they are sourcable shell scripts which primarily consist of environment variable assignments. Most of the files in these directories are sourced by the SysV init scripts of the same name. The Debian Policy Manual (9.3.2) and the Fedora Packaging Guidelines suggest this use of the directories, however both distributions also have files in them that do not follow this scheme, i.e. that do not have a matching SysV init script -- or not even are shell scripts at all.

Why have these files been introduced? On SysV systems services are started via init scripts in `/etc/rc.d/init.d` (or a similar directory). `/etc/` is (these days) considered the place where system configuration is stored. Originally these init scripts were subject to customization by the administrator. But as they grew and become complex most distributions no longer considered them true configuration files, but more just a special kind of programs. To make customization easy and guarantee a safe upgrade path the customizable bits hence have been moved to separate configuration files, which the init scripts then source.

Let's have a quick look what kind of configuration you can do with these files. Here's a short incomprehensive list of various things that can be configured via environment settings in these source files I found browsing through the directories on a Fedora and a Debian machine:

- Additional command line parameters for the daemon binaries
- Locale settings for a daemon
- Shutdown time-out for a daemon
- Shutdown mode for a daemon
- System configuration like system locale, time zone information, console keyboard
- Redundant system configuration, like whether the RTC is in local timezone
- Firewall configuration data, not in shell format (!)
- CPU affinity for a daemon
- Settings unrelated to boot, for example including information how to install a new kernel package, how to configure nspluginwrap or whether to do library prelinking
- Whether a specific service should be started or not
- Networking configuration
- Which kernel modules to statically load
- Whether to halt or power-off on shutdown
- Access modes for device nodes (!)
- A description string for the SysV service (!)
- The user/group ID, umask to run specific daemons as
- Resource limits to set for a specific daemon
- OOM adjustment to set for a specific daemon

Now, let's go where the beef is: what's wrong with `/etc/sysconfig` (resp. `/etc/default`)? Why might it make sense to fade out use of these files in a systemd world?

- For the majority of these files the reason for having them simply does not exist anymore: systemd unit files are not programs like SysV init scripts were. Unit files are simple, declarative descriptions, that usually do not consist of more than 6 lines or so. They can easily be generated, parsed without a Bourne interpreter and understood by the reader. Also, they are very easy to modify: just copy them from `/lib/systemd/system` to `/etc/systemd/system` and edit them there, where they will not be modified by the package manager. The need to separate code and configuration that was the original reason to introduce these files does not exist anymore, as systemd unit files do not include code. These files hence now are a solution looking for a problem that no longer exists.
- They are inherently distribution-specific. With systemd we hope to encourage standardization between distributions. Part of this is that we want that unit files are supplied with upstream, and not just added by the packager -- how it has usually been done in the SysV world. Since the location of the directory and the available variables in the files is very different on each distribution, supporting `/etc/sysconfig` files in upstream unit files is not feasible. Configuration stored in these files works against de-balkanization of the Linux platform.
- Many settings are fully redundant in a systemd world. For example, various services support configuration of the process credentials like the user/group ID, resource limits, CPU affinity or the OOM adjustment settings. However, these settings are supported only by some SysV init scripts, and often have different names if supported in multiple of them. OTOH in systemd, all these settings are available equally and uniformly for all services, with the same configuration option in unit files.
- Unit files know a large number of easy-to-use process context settings, that are more comprehensive than what most `/etc/sysconfig` files offer.
- A number of these settings are entirely questionnabe. For example, the aforementiond configuration option for the user/group ID a service runs as is primarily something the distributor has to take care of. There is little to win for administrators to change these settings, and only the distributor has the broad overview to make sure that UID/GID and name

collisions do not happen.

- The file format is not ideal. Since the files are usually sourced as shell scripts, parse errors are very hard to decypher and are not logged along the other configuration problems of the services. Generally, unknown variable assignments simply have no effect but this is not warned about. This makes these files harder to debug than necessary.
- Configuration files sources from shell scripts are subject to the execution parameters of the interpreter, and it has many: settings like IFS or LANG tend to modify drastically how shell scripts are parsed and understood. This makes them fragile.
- Interpretation of these files is slow, since it requires spawning of a shell, which adds at least one process for each service to be spawned at boot.
- Often, files in `/etc/sysconfig` are used to "fake" configuration files for daemons which do not support configuration files natively. This is done by glueing together command line arguments from these variable assignments that are then passed to the daemon. In general proper, native configuration files in these daemons are the much prettier solution however. Command line options like "-k", "-a" or "-f" are not self-explanatory and have a very cryptic syntax. Moreover the same switches in many daemons have (due to the limited vocabulary) often very much contradicting effects. (On one daemon `–f` might cause the daemon to daemonize, while on another one this option turns exactly this behaviour off.) Command lines generally cannot include sensible comments which most configuration files however can.
- A number of configuration settings in `/etc/sysconfig` are entirely redundant: for example, on many distributions it can be controlled via `/etc/sysconfig` files whether the RTC is in UTC or local time. Such an option already exists however in the 3rd line of the `/etc/adjtime` (which is known on all distributions). Adding a second, redundant, distribution-specific option overriding this is hence needless and complicates things for no benefit.
- Many of the configuration settings in `/etc/sysconfig` allow disabling services. By this they basically become a second level of enabling/disabling over what the init system already offers: when a service is enabled with `systemctl enable` or `chkconfig on` these settings override this, and turn the daemon of even though the init system was configured to start it. This of

course is very confusing to the user/administrator, and brings virtually no benefit.
- For options like the configuration of static kernel modules to load: there are nowadays usually much better ways to load kernel modules at boot. For example, most modules may now be autoloaded by udev when the right hardware is found. This goes very far, and even includes ACPI and other high-level technologies. One of the very few exceptions where we currently do not do kernel module autoloading is CPU feature and model based autoloading which however will be supported soon too. And even if your specific module cannot be auto-loaded there's usually a better way to statically load it, for example by sticking it in `/etc/load-modules.d` so that the administrator can check a standardized place for all statically loaded modules.
- Last but not least, /etc already is intended to be the place for system configuration ("Host-specific system configuration" according to FHS). A subdirectory beneath it called `sysconfig` to place system configuration in is hence entirely redundant, already on the language level.

What to use instead? Here are a few recommendations of what to do with these files in the long run in a systemd world:

- Just drop them without replacement. If they are fully redundant (like the local/UTC RTC setting) this is should be a relatively easy way out (well, ignoring the need for compatibility). If systemd natively supports an equivalent option in the unit files there is no need to duplicate these settings in `sysconfig` files. For a list of execution options you may set for a service check out the respective man pages: systemd.exec(5) and systemd.service(5). If your setting simply adds another layer where a service can be disabled, remove it to keep things simple. There's no need to have multiple ways to disable a service.
- Find a better place for them. For configuration of the system locale or system timezone we hope to gently push distributions into the right direction, for more details see previous episode of this series.
- Turn these settings into native settings of the daemon. If necessary add support for reading native configuration files to the daemon. Thankfully, most of the stuff we run on Linux is Free Software, so this can relatively easily be done.

Of course, there's one very good reason for supporting these files for a bit longer: compatibility for upgrades. But that's is really the only one I could come up with. It's reason enough to keep compatibility for a while, but I think it is a good idea to phase out usage of these files at least in new packages.

If compatibility is important, then systemd will still allow you to read these configuration files even if you otherwise use native systemd unit files. If your `sysconfig` file only knows simple options `EnvironmentFile=-/etc/sysconfig/foobar` (See systemd.exec(5) for more information about this option.) may be used to import the settings into the environment and use them to put together command lines. If you need a programming language to make sense of these settings, then use a programming language like shell. For example, place an short shell script in `/usr/lib/`*`<your package>`*`/` which reads these files for compatibility, and then `exec`'s the actual daemon binary. Then spawn this script instead of the actual daemon binary with `ExecStart=` in the unit file.

And this is all for now. Thank you very much for your interest.

systemd for Administrators, Part X

Here's the tenth installment of my ongoing series on systemd for Administrators:

Instantiated Services

Most services on Linux/Unix are *singleton* services: there's usually only one instance of Syslog, Postfix, or Apache running on a specific system at the same time. On the other hand some select services may run in multiple instances on the same host. For example, an Internet service like the Dovecot IMAP service could run in multiple instances on different IP ports or different local IP addresses. A more common example that exists on all installations is *getty*, the mini service that runs once for each TTY and presents a login prompt on it. On most systems this service is instantiated once for each of the first six virtual consoles `tty1` to `tty6`. On some servers depending on administrator configuration or boot-time parameters an additional getty is instantiated for a serial or virtualizer console. Another common instantiated service in the systemd world is *fsck*, the file system checker that is instantiated once for each block device that needs to be checked. Finally, in systemd socket activated per-connection services (think classic inetd!) are also implemented via instantiated services: a new instance is created for each incoming connection. In this installment I hope to explain a bit how systemd implements instantiated services and how to take advantage of them as an administrator.

If you followed the previous episodes of this series you are probably aware that services in systemd are named according to the pattern *`foobar`*`.service`, where *foobar* is an identification string for the service, and `.service` simply a fixed suffix that is identical for all service units. The definition files for these services are searched for in `/etc/systemd/system` and `/lib/systemd/system` (and possibly other directories) under this name. For instantiated services this pattern is extended a bit: the service name becomes *`foobar@quux`*`.service` where *foobar* is the common service identifier, and *quux* the instance identifier. Example: `serial-getty@ttyS2.service` is the serial getty service instantiated for `ttyS2`.

Service instances can be created dynamically as needed. Without further configuration you may easily start a new getty on a serial port simply by invoking a `systemctl start` command for the new instance:

```
# systemctl start serial-getty@ttyUSB0.service
```

If a command like the above is run systemd will first look for a unit configuration file by the exact name you requested. If this service file is not found (and usually it isn't if you use instantiated services like this) then the instance id is removed from the name and a unit configuration file by the resulting *template* name searched. In other words, in the above example, if the precise `serial-getty@ttyUSB0.service` unit file cannot be found, `serial-getty@.service` is loaded instead. This unit template file will hence be common for all instances of this service. For the serial getty we ship a template unit file in systemd (`/lib/systemd/system/serial-getty@.service`) that looks something like this:

```
[Unit]
Description=Serial Getty on %I
BindTo=dev-%i.device
After=dev-%i.device systemd-user-sessions.service

[Service]
ExecStart=-/sbin/agetty -s %I 115200,38400,9600
Restart=always
RestartSec=0
```

(Note that the unit template file we actually ship along with systemd for the serial gettys is a bit longer. If you are interested, have a look at the actual file which includes additional directives for compatibility with SysV, to clear the screen and remove previous users from the TTY device. To keep things simple I have shortened the unit file to the relevant lines here.)

This file looks mostly like any other unit file, with one distinction: the specifiers `%I` and `%i` are used at multiple locations. At unit load time `%I` and `%i` are replaced by systemd with the instance identifier of the service. In our example above, if a service is instantiated as `serial-getty@ttyUSB0.service` the specifiers `%I` and `%i` will be replaced by `ttyUSB0`. If you introspect the instanciated unit with `systemctl status serial-getty@ttyUSB0.service` you will see these replacements having taken place:

```
$ systemctl status serial-getty@ttyUSB0.service
serial-getty@ttyUSB0.service - Getty on ttyUSB0
   Loaded: loaded (/lib/systemd/system/serial-getty@.service; static)
   Active: active (running) since Mon, 26 Sep 2011 04:20:44 +0200; 2s ago
 Main PID: 5443 (agetty)
   CGroup: name=systemd:/system/getty@.service/ttyUSB0
     └ 5443 /sbin/agetty -s ttyUSB0 115200,38400,9600
```

And that is already the core idea of instantiated services in systemd. As you can see systemd provides a very simple templating system, which can be used to dynamically instantiate services as needed. To make effective use of this, a few more notes:

You may instantiate these services *on-the-fly* in `.wants/` symbolic links in the file system. For example, to make sure the serial getty on `ttyUSB0` is started automatically at every boot, create a symlink like this:

```
# ln -s /lib/systemd/system/serial-getty@.service /etc/systemd/system/getty.target.wa
```

systemd will instantiate the symlinked unit file with the instance name specified in the symlink name.

You cannot instantiate a unit template without specifying an instance identifier. In other words `systemctl start serial-getty@.service` will necessarily fail since the instance name was left unspecified.

Sometimes it is useful to *opt-out* of the generic template for one specific instance. For these cases make use of the fact that systemd always searches first for the full instance file name before falling back to the template file name: make sure to place a unit file under the fully instantiated name in `/etc/systemd/system` and it will override the generic templated version for this specific instance.

The unit file shown above uses `%i` at some places and `%I` at others. You may wonder what the difference between these specifiers are. `%i` is replaced by the exact characters of the instance identifier. For `%I` on the other hand the instance identifier is first passed through a simple unescaping algorithm. In the case of a simple instance identifier like `ttyUSB0` there is no effective difference. However, if the device name includes one or more slashes ("`/`") this cannot be part of a unit name (or Unix file name). Before such a device name can be used as instance identifier it needs to be escaped so that "/" becomes "-" and most other special characters (including "-") are replaced by "\xAB" where AB is the ASCII code of the character in hexadecimal notation[1]. Example: to refer to a USB serial port by its bus path we want to use a port name like `serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0`. The escaped version of this name is `serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0`. `%I` will then refer to former, `%i` to the latter. Effectively this means `%i` is useful wherever it is necessary to refer to other units, for example to express additional dependencies. On the other hand `%I` is useful for usage in command lines, or inclusion in pretty description strings. Let's check how this looks with the above unit file:

```
# systemctl start 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.
# systemctl status 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:
serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0.se
   Loaded: loaded (/lib/systemd/system/serial-getty@.service; static)
   Active: active (running) since Mon, 26 Sep 2011 05:08:52 +0200; 1s ago
 Main PID: 5788 (agetty)
   CGroup: name=systemd:/system/serial-getty@.service/serial-by\x2dpath-pci\x2d000
    └ 5788 /sbin/agetty -s serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0 115200 
```

As we can see the while the instance identifier is the escaped string the command line and the description string actually use the unescaped version, as expected.

(Side note: there are more specifiers available than just `%i` and `%I`, and many of them are actually available in all unit files, not just templates for service instances. For more details see the man page which includes a full list and terse explanations.)

And at this point this shall be all for now. Stay tuned for a follow-up article on how instantiated services are used for `inetd`-style socket activation.

Footnotes

[1] Yupp, this escaping algorithm doesn't really result in particularly pretty escaped strings, but then again, most escaping algorithms don't help readability. The algorithm we used here is inspired by what udev does in a similar case, with one change. In the end, we had to pick something. If you'll plan to comment on the escaping algorithm please also mention where you live so that I can come around and paint your bike shed yellow with blue stripes. Thanks!

systemd for Administrators, Part XI

Here's the eleventh installment of my ongoing series on systemd for Administrators:

Converting inetd Services

In a previous episode of this series I covered how to convert a SysV init script to a systemd unit file. In this story I hope to explain how to convert inetd services into systemd units.

Let's start with a bit of background. inetd has a long tradition as one of the classic Unix services. As a superserver it listens on an Internet socket on behalf of another service and then activate that service on an incoming connection, thus implementing an on-demand socket activation system. This allowed Unix machines with limited resources to provide a large variety of services, without the need to run processes and invest resources for all of them all of the time. Over the years a number of independent implementations of inetd have been shipped on Linux distributions. The most prominent being the ones based on BSD inetd and xinetd. While inetd used to be installed on most distributions by default, it nowadays is used only for very few selected services and the common services are all run unconditionally at boot, primarily for (perceived) performance reasons.

One of the core feature of systemd (and Apple's launchd for the matter) is socket activation, a scheme pioneered by inetd, however back then with a different focus. Systemd-style socket activation focusses on local sockets (AF_UNIX), not so much Internet sockets (AF_INET), even though both are supported. And more importantly even, socket activation in systemd is not primarily about the on-demand aspect that was key in inetd, but more on increasing parallelization (socket activation allows starting clients and servers of the socket at the same time), simplicity (since the need to configure explicit dependencies between services is removed) and robustness (since services can be restarted or may crash without loss of connectivity of the socket). However, systemd can also activate services on-demand when connections are incoming, if configured that way.

Socket activation of any kind requires support in the services themselves. systemd provides a very simple interface that services may implement to provide socket activation, built around sd_listen_fds(). As such it is already a very minimal, simple scheme. However, the traditional inetd interface is even simpler. It allows passing only a single socket to the activated service: the socket fd is simply duplicated to STDIN and STDOUT of the process spawned, and that's already it. In order to provide compatibility systemd optionally offers the same interface to processes, thus taking advantage of the many services that already support inetd-style socket activation, but not yet systemd's native activation.

Before we continue with a concrete example, let's have a look at three different schemes to make use of socket activation:

1. **Socket activation for parallelization, simplicity, robustness:** sockets are bound during early boot and a singleton service instance to serve all client requests is immediately started at boot. This is useful for all services that are very likely used frequently and continously, and hence starting them early and in parallel with the rest of the system is advisable. Examples: D-Bus, Syslog.
2. **On-demand socket activation for singleton services:** sockets are bound during early boot and a singleton service instance is executed on incoming traffic. This is useful for services that are seldom used, where it is advisable to save the resources and time at boot and delay activation until they are actually needed. Example: CUPS.
3. **On-demand socket activation for per-connection service instances:** sockets are bound during early boot and for each incoming connection a new service instance is instantiated and the connection socket (and not the listening one) is passed to it. This is useful for services that are seldom used, and where performance is not critical, i.e. where the cost of spawning a new service process for each incoming connection is limited. Example: SSH.

The three schemes provide different performance characteristics. After the service finishes starting up the performance provided by the first two schemes is identical to a stand-alone service (i.e. one that is started without a super-server, without socket activation), since the listening socket is passed to the actual service, and code paths from then on are identical to those of a stand-alone service and all connections are processes exactly the same way as they are in a stand-alone service. On the other hand, performance of the third scheme is usually not as good: since for each connection a new service needs to be started the resource cost is much higher. However, it also has a number of advantages: for example client connections are better isolated and it is easier to develop services activated this way.

For systemd primarily the first scheme is in focus, however the other two schemes are supported as well. (In fact, the blog story I covered the necessary code changes for systemd-style socket activation in was about a service of the second type, i.e. CUPS). inetd primarily focusses on the third scheme, however the second scheme is supported too. (The first one isn't. Presumably due the focus on the third scheme inetd got its -- a bit unfair -- reputation for being "slow".)

So much about the background, let's cut to the beef now and show an inetd service can be integrated into systemd's socket activation. We'll focus on SSH, a very common service that is widely installed and used but on the vast majority of machines probably not started more often than 1/h in average (and usually even much less). SSH has supported inetd-style activation since a long time, following the third scheme mentioned above. Since it is started only every now and then and only with a limited number of connections at the same time it is a very good candidate for this scheme as the extra resource cost is negligble: if made socket-activatable SSH is basically free as long as nobody uses it. And as soon as somebody logs in via SSH it will be started and the moment he or she disconnects all its resources are freed again. Let's find out how to make SSH socket-activatable in systemd taking advantage of the provided inetd compatibility!

Here's the configuration line used to hook up SSH with classic inetd:

```
ssh stream tcp nowait root /usr/sbin/sshd sshd -i
```

And the same as xinetd configuration fragment:

```
service ssh {
        socket_type = stream
        protocol = tcp
        wait = no
        user = root
        server = /usr/sbin/sshd
        server_args = -i
}
```

Most of this should be fairly easy to understand, as these two fragments express very much the same information. The non-obvious parts: the port number (22) is not configured in inetd configuration, but indirectly via the service database in `/etc/services`: the service name is used as lookup key in that database and translated to a port number. This indirection via `/etc/services` has been part of Unix tradition though has been getting more and more out of fashion, and the newer xinetd hence optionally allows configuration with explicit port numbers. The most interesting setting here is the not very intuitively named `nowait` (resp. `wait=no`) option. It configures whether a service is of the second (`wait`) resp. third (`nowait`) scheme mentioned above. Finally the `-i` switch is used to enabled inetd mode in SSH.

The systemd translation of these configuration fragments are the following two units. First: `sshd.socket` is a unit encapsulating information about a socket to listen on:

```
[Unit]
Description=SSH Socket for Per-Connection Servers

[Socket]
ListenStream=22
Accept=yes

[Install]
WantedBy=sockets.target
```

Most of this should be self-explanatory. A few notes: `Accept=yes` corresponds to `nowait`. It's hopefully better named, referring to the fact that for `nowait` the superserver calls `accept()` on the listening socket, where for `wait` this is the job of the executed service process. `WantedBy=sockets.target` is used to ensure that when enabled this unit is activated at boot at the right time.

And here's the matching service file `sshd@.service`:

```
[Unit]
Description=SSH Per-Connection Server

[Service]
ExecStart=-/usr/sbin/sshd -i
StandardInput=socket
```

This too should be mostly self-explanatory. Interesting is `StandardInput=socket`, the option that enables inetd compatibility for this service. `StandardInput=` may be used to configure what STDIN of the service should be connected for this service (see the man page for details). By setting it to `socket` we make sure to pass the connection socket here, as expected in the simple inetd interface. Note that we do not need to explicitly configure `StandardOutput=` here, since by default the setting from `StandardInput=` is inherited if nothing else is configured. Important is the "-" in front of the binary name. This ensures that the exit status of the per-connection sshd process is forgotten by systemd. Normally, systemd will store the exit status of a all service instances that die abnormally. SSH will sometimes die abnormally with an exit code of 1 or similar, and we want to make sure that this doesn't cause systemd to keep around information for numerous previous connections that died this way (until this information is forgotten with `systemctl reset-failed`).

`sshd@.service` is an instantiated service, as described in the preceeding installment of this series. For each incoming connection systemd will instantiate a new instance of `sshd@.service`, with the instance identifier named after the connection credentials.

You may wonder why in systemd configuration of an inetd service requires two unit files instead of one. The reason for this is that to simplify things we want to make sure that the relation between live units and unit files is obvious, while at the same time we can order the socket unit and the service units independently in the dependency graph and control the units as independently as possible. (Think: this allows you to shutdown the socket independently from the instances, and each instance individually.)

Now, let's see how this works in real life. If we drop these files into `/etc/systemd/system` we are ready to enable the socket and start it:

```
# systemctl enable sshd.socket
ln -s '/etc/systemd/system/sshd.socket' '/etc/systemd/system/sockets.target.wants/ss
# systemctl start sshd.socket
# systemctl status sshd.socket
sshd.socket - SSH Socket for Per-Connection Servers
   Loaded: loaded (/etc/systemd/system/sshd.socket; enabled)
   Active: active (listening) since Mon, 26 Sep 2011 20:24:31 +0200; 14s ago
 Accepted: 0; Connected: 0
   CGroup: name=systemd:/system/sshd.socket
```

This shows that the socket is listening, and so far no connections have been made (`Accepted:` will show you how many connections have been made in total since the socket was started, `Connected:` how many connections are currently active.)

Now, let's connect to this from two different hosts, and see which services are now active:

```
$ systemctl --full | grep ssh
sshd@172.31.0.52:22-172.31.0.4:47779.service  loaded active running      SSH Per-Co
sshd@172.31.0.52:22-172.31.0.54:52985.service loaded active running      SSH Per-C
sshd.socket                                   loaded active listening    SSH Socket for Per-Connecti
```

As expected, there are now two service instances running, for the two connections, and they are named after the source and destination address of the TCP connection as well as the port numbers. (For AF_UNIX sockets the instance identifier will carry the PID and UID of the connecting client.) This allows us to invidiually introspect or kill specific sshd instances, in case you want to terminate the session of a specific client:

```
# systemctl kill sshd@172.31.0.52:22-172.31.0.4:47779.service
```

And that's probably already most of what you need to know for hooking up inetd services with systemd and how to use them afterwards.

In the case of SSH it is probably a good suggestion for most distributions in order to save resources to default to this kind of inetd-style socket activation, but provide a stand-alone unit file to sshd as well which can be enabled optionally. I'll soon file a wishlist bug about this against our SSH package in Fedora.

A few final notes on how xinetd and systemd compare feature-wise, and whether xinetd is fully obsoleted by systemd. The short answer here is that systemd does not provide the full xinetd feature set and that is does not fully obsolete xinetd. The longer answer is a bit more complex: if you look at the multitude of options xinetd provides you'll notice that systemd does not compare. For example, systemd does not come with built-in `echo`, `time`, `daytime` or `discard` servers, and never will include those. TCPMUX is not supported, and neither are RPC services. However, you will also find that most of these are either irrelevant on today's Internet or became other way out-of-fashion. The vast majority of inetd services do not directly take advantage of these additional features. In fact, none of the xinetd services shipped on Fedora make use of these options. That said, there are a couple of useful features that systemd does not support, for example IP ACL management. However, most administrators will probably agree that firewalls are the better solution for these kinds of problems and on top of that, systemd supports ACL management via tcpwrap for those who indulge in retro technologies like this. On the other hand systemd also provides numerous features `xinetd` does not provide, starting with the individual control of instances shown above, or the more expressive configurability of the execution context for the instances. I believe that what systemd provides is quite comprehensive, comes with little legacy cruft but should provide you with everything you need. And if there's something systemd does not cover, `xinetd` will always be there to fill the void as you can easily run it in conjunction with `systemd`. For the majority of uses systemd should cover what is necessary, and allows you cut down on the required components to build your system from. In a way, systemd brings back the functionality of classic Unix inetd and turns it again into a center piece of a Linux system.

And that's all for now. Thanks for reading this long piece. And now, get going and convert your services over! Even better, do this work in the individual packages upstream or in your distribution!

systemd for Administrators, Part XII

Here's the twelfth installment of my ongoing series on systemd for Administrators:

Securing Your Services

One of the core features of Unix systems is the idea of privilege separation between the different components of the OS. Many system services run under their own user IDs thus limiting what they can do, and hence the impact they may have on the OS in case they get exploited.

This kind of privilege separation only provides very basic protection however, since in general system services run this way can still do at least as much as a normal local users, though not as much as root. For security purposes it is however very interesting to limit even further what services can do, and shut them off a couple of things that normal users are allowed to do.

A great way to limit the impact of services is by employing MAC technologies such as SELinux. If you are interested to secure down your server, running SELinux is a very good idea. systemd enables developers and administrators to apply additional restrictions to local services independently of a MAC. Thus, regardless whether you are able to make use of SELinux you may still enforce certain security limits on your services.

In this iteration of the series we want to focus on a couple of these security features of systemd and how to make use of them in your services. These features take advantage of a couple of Linux-specific technologies that have been available in the kernel for a long time, but never have been exposed in a widely usable fashion. These systemd features have been designed to be as easy to use as possible, in order to make them attractive to administrators and upstream developers:

- Isolating services from the network
- Service-private `/tmp`
- Making directories appear read-only or inaccessible to services
- Taking away capabilities from services

- Disallowing forking, limiting file creation for services
- Controlling device node access of services

All options described here are documented in systemd's man pages, notably systemd.exec(5). Please consult these man pages for further details.

All these options are available on all systemd systems, regardless if SELinux or any other MAC is enabled, or not.

All these options are relatively cheap, so if in doubt use them. Even if you might think that your service doesn't write to `/tmp` and hence enabling `PrivateTmp=yes` (as described below) might not be necessary, due to today's complex software it's still beneficial to enable this feature, simply because libraries you link to (and plug-ins to those libraries) which you do not control might need temporary files after all. Example: you never know what kind of NSS module your local installation has enabled, and what that NSS module does with `/tmp`.

These options are hopefully interesting both for administrators to secure their local systems, and for upstream developers to ship their services secure by default. We strongly encourage upstream developers to consider using these options by default in their upstream service units. They are very easy to make use of and have major benefits for security.

Isolating Services from the Network

A very simple but powerful configuration option you may use in systemd service definitions is `PrivateNetwork=`:

```
...
[Service]
ExecStart=...
PrivateNetwork=yes
...
```

With this simple switch a service and all the processes it consists of are entirely disconnected from any kind of networking. Network interfaces became unavailable to the processes, the only one they'll see is the loopback device "lo", but it is isolated from the real host loopback. This is a very powerful protection from network attacks.

Caveat: Some services require the network to be operational. Of course, nobody would consider using `PrivateNetwork=yes` on a network-facing service such as Apache. However even for non-network-facing services network support might be necessary and not always obvious. Example: if the local system is configured for an LDAP-based user database doing glibc name lookups with calls such as `getpwnam()` might end up resulting in network access. That said, even in those cases it is more often than not OK to use `PrivateNetwork=yes` since user IDs of system service users are required to be resolvable even without any network around. That means as long as the only user IDs your service needs to resolve are below the magic 1000 boundary using `PrivateNetwork=yes` should be OK.

Internally, this feature makes use of network namespaces of the kernel. If enabled a new network namespace is opened and only the loopback device configured in it.

Service-Private /tmp

Another very simple but powerful configuration switch is `PrivateTmp=`:

```
...
[Service]
ExecStart=...
PrivateTmp=yes
...
```

If enabled this option will ensure that the `/tmp` directory the service will see is private and isolated from the host system's `/tmp`. `/tmp` traditionally has been a shared space for all local services and users. Over the years it has been a major source of security problems for a multitude of services. Symlink attacks and DoS vulnerabilities due to guessable `/tmp` temporary files are common. By isolating the service's `/tmp` from the rest of the host, such vulnerabilities become moot.

For Fedora 17 a feature has been accepted in order to enable this option across a large number of services.

Caveat: Some services actually misuse `/tmp` as a location for IPC sockets and other communication primitives, even though this is almost always a vulnerability (simply because if you use it for communication you need guessable names, and guessable names make your code vulnerable to DoS and symlink attacks) and `/run` is the much safer replacement for this, simply because it is not a location writable to unprivileged processes. For example, X11 places it's communication sockets below `/tmp` (which is actually secure -- though still not ideal -- in this exception since it does so in a safe subdirectory which is created at early boot.) Services which need to communicate via such communication primitives in `/tmp` are no candidates for `PrivateTmp=`. Thankfully these days only very few services misusing `/tmp` like this remain.

Internally, this feature makes use of file system namespaces of the kernel. If enabled a new file system namespace is opened inheritng most of the host hierarchy with the exception of `/tmp`.

Making Directories Appear Read-Only or Inaccessible to Services

With the `ReadOnlyDirectories=` and `InaccessibleDirectories=` options it is possible to make the specified directories inaccessible for writing resp. both reading and writing to the service:

```
...
[Service]
ExecStart=...
InaccessibleDirectories=/home
ReadOnlyDirectories=/var
...
```

With these two configuration lines the whole tree below `/home` becomes inaccessible to the service (i.e. the directory will appear empty and with 000 access mode), and the tree below `/var` becomes read-only.

Caveat: Note that `ReadOnlyDirectories=` currently is not recursively applied to submounts of the specified directories (i.e. mounts below `/var` in the example above stay writable). This is likely to get fixed soon.

Internally, this is also implemented based on file system namspaces.

Taking Away Capabilities From Services

Another very powerful security option in systemd is `CapabilityBoundingSet=` which allows to limit in a relatively fine grained fashion which kernel capabilities a service started retains:

```
...
[Service]
ExecStart=...
CapabilityBoundingSet=CAP_CHOWN CAP_KILL
...
```

In the example above only the CAP_CHOWN and CAP_KILL capabilities are retained by the service, and the service and any processes it might create have no chance to ever acquire any other capabilities again, not even via setuid binaries. The list of currently defined capabilities is available in capabilities(7). Unfortunately some of the defined capabilities are overly generic (such as CAP_SYS_ADMIN), however they are still a very useful tool, in particular for services that otherwise run with full root privileges.

To identify precisely which capabilities are necessary for a service to run cleanly is not always easy and requires a bit of testing. To simplify this process a bit, it is possible to blacklist certain capabilities that are definitely not needed instead of whitelisting all that might be needed. Example: the CAP_SYS_PTRACE is a particularly powerful and security relevant capability needed for the implementation of debuggers, since it allows introspecting and manipulating any local process on the system. A service like Apache obviously has no business in being a debugger for other processes, hence it is safe to remove the capability from it:

```
...
[Service]
ExecStart=...
CapabilityBoundingSet=~CAP_SYS_PTRACE
...
```

The ~ character the value assignment here is prefixed with inverts the meaning of the option: instead of listing all capabalities the service will retain you may list the ones it will not retain.

Caveat: Some services might react confused if certain capabilities are made unavailable to them. Thus when determining the right set of capabilities to keep around you need to do this carefully, and it might be a good idea to talk to the upstream maintainers since they should know best which operations a service might need to run successfully.

Caveat 2: Capabilities are not a magic wand. You probably want to combine them and use them in conjunction with other security options in order to make them truly useful.

To easily check which processes on your system retain which capabilities use the `pscap` tool from the `libcap-ng-utils` package.

Making use of systemd's `CapabilityBoundingSet=` option is often a simple, discoverable and cheap replacement for patching all system daemons individually to control the capability bounding set on their own.

Disallowing Forking, Limiting File Creation for Services

Resource Limits may be used to apply certain security limits on services being run. Primarily, resource limits are useful for resource control (as the name suggests...) not so much access control. However, two of them can be useful to disable certain OS features: RLIMIT_NPROC and RLIMIT_FSIZE may be used to disable forking and disable writing of any files with a size > 0:

```
...
[Service]
ExecStart=...
LimitNPROC=1
LimitFSIZE=0
...
```

Note that this will work only if the service in question drops privileges and runs under a (non-root) user ID of its own or drops the CAP_SYS_RESOURCE capability, for example via `CapabilityBoundingSet=` as discussed above. Without that a process could simply increase the resource limit again thus voiding any effect.

Caveat: `LimitFSIZE=` is pretty brutal. If the service attempts to write a file with a size > 0, it will immeidately be killed with the SIGXFSZ which unless caught terminates the process. Also, creating files with size 0 is still allowed, even if this option is used.

For more information on these and other resource limits, see setrlimit(2).

Controlling Device Node Access of Services

Devices nodes are an important interface to the kernel and its drivers. Since drivers tend to get much less testing and security checking than the core kernel they often are a major entry point for security hacks. systemd allows you to control access to devices individually for each service:

```
...
[Service]
ExecStart=...
DeviceAllow=/dev/null rw
...
```

This will limit access to `/dev/null` and only this device node, disallowing access to any other device nodes.

The feature is implemented on top of the `devices` cgroup controller.

Other Options

Besides the easy to use options above there are a number of other security relevant options available. However they usually require a bit of preparation in the service itself and hence are probably primarily useful for upstream developers. These options are `RootDirectory=` (to set up `chroot()` environments for a service) as well as `User=` and `Group=` to drop privileges to the specified user and group. These options are particularly useful to greatly simplify writing daemons, where all the complexities of securely dropping privileges can be left to systemd, and kept out of the daemons themselves.

If you are wondering why these options are not enabled by default: some of them simply break seamntics of traditional Unix, and to maintain compatibility we cannot enable them by default. e.g. since traditional Unix enforced that `/tmp` was a shared namespace, and processes could use it for IPC we cannot just go and turn that off globally, just because `/tmp`'s role in IPC is now replaced by `/run`.

And that's it for now. If you are working on unit files for upstream or in your distribution, please consider using one or more of the options listed above. If you service is secure by default by taking advantage of these options this will help not only your users but also make the Internet a safer place.

systemd for Administrators, Part XIII

Here's the thirteenth installment of my ongoing series on systemd for Administrators:

Log and Service Status

This one is a short episode. One of the most commonly used commands on a systemd system is `systemctl status` which may be used to determine the status of a service (or other unit). It always has been a valuable tool to figure out the processes, runtime information and other meta data of a daemon running on the system.

With Fedora 17 we introduced the journal, our new logging scheme that provides structured, indexed and reliable logging on systemd systems, while providing a certain degree of compatibility with classic syslog implementations. The original reason we started to work on the journal was one specific feature idea, that to the outsider might appear simple but without the journal is difficult and inefficient to implement: along with the output of `systemctl status` we wanted to show the last 10 log messages of the daemon. Log data is some of the most essential bits of information we have on the status of a service. Hence it it is an obvious choice to show next to the general status of the service.

And now to make it short: at the same time as we integrated the journal into `systemd` and Fedora we also hooked up `systemctl` with it. Here's an example output:

```
$ systemctl status avahi-daemon.service
avahi-daemon.service - Avahi mDNS/DNS-SD Stack
   Loaded: loaded (/usr/lib/systemd/system/avahi-daemon.service; enabled)
   Active: active (running) since Fri, 18 May 2012 12:27:37 +0200; 14s ago
 Main PID: 8216 (avahi-daemon)
   Status: "avahi-daemon 0.6.30 starting up."
   CGroup: name=systemd:/system/avahi-daemon.service
     ├ 8216 avahi-daemon: running [omega.local]
     └ 8217 avahi-daemon: chroot helper

May 18 12:27:37 omega avahi-daemon[8216]: Joining mDNS multicast group on interfa
May 18 12:27:37 omega avahi-daemon[8216]: New relevant interface eth1.IPv4 for mD
May 18 12:27:37 omega avahi-daemon[8216]: Network interface enumeration complet
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for 192
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for fd0
May 18 12:27:37 omega avahi-daemon[8216]: Registering new address record for 172
May 18 12:27:37 omega avahi-daemon[8216]: Registering HINFO record with values 'X
May 18 12:27:38 omega avahi-daemon[8216]: Server startup complete. Host name is
May 18 12:27:38 omega avahi-daemon[8216]: Service "omega" (/services/ssh.service)
May 18 12:27:38 omega avahi-daemon[8216]: Service "omega" (/services/sftp-ssh.ser
```

This, of course, shows the status of everybody's favourite mDNS/DNS-SD daemon with a list of its processes, along with -- as promised -- the 10 most recent log lines. Mission accomplished!

There are a couple of switches available to alter the output slightly and adjust it to your needs. The two most interesting switches are `-f` to enable follow mode (as in `tail -f`) and `-n` to change the number of lines to show (you guessed it, as in `tail -n`).

The log data shown comes from three sources: everything any of the daemon's processes logged with libc's `syslog()` call, everything submitted using the native Journal API, plus everything any of the daemon's processes logged to STDOUT or STDERR. In short: everything the daemon generates as log data is collected, properly interleaved and shown in the same format.

And that's it already for today. It's a very simple feature, but an immensely useful one for every administrator. One of the kind "Why didn't we already do this 15 years ago?".

Stay tuned for the next installment!

systemd for Administrators, Part XIV

And here's the fourteenth installment of my ongoing series on systemd for Administrators:

The Self-Explanatory Boot

One complaint we often hear about systemd is that its boot process was hard to understand, even incomprehensible. In general I can only disagree with this sentiment, I even believe in quite the opposite: in comparison to what we had before -- where to even remotely understand what was going on you had to have a decent comprehension of the programming language that is Bourne Shell[1] -- understanding systemd's boot process is substantially easier. However, like in many complaints there is some truth in this frequently heard discomfort: for a seasoned Unix administrator there indeed is a bit of learning to do when the switch to systemd is made. And as systemd developers it is our duty to make the learning curve shallow, introduce as few surprises as we can, and provide good documentation where that is not possible.

systemd always had huge body of documentation as manual pages (nearly 100 individual pages now!), in the Wiki and the various blog stories I posted. However, any amount of documentation alone is not enough to make software easily understood. In fact, thick manuals sometimes appear intimidating and make the reader wonder where to start reading, if all he was interested in was this one simple concept of the whole system.

Acknowledging all this we have now added a new, neat, little feature to systemd: the self-explanatory boot process. What do we mean by that? Simply that each and every single component of our boot comes with documentation and that this documentation is closely linked to its component, so that it is easy to find.

More specifically, all units in systemd (which are what encapsulate the components of the boot) now include references to their documentation, the documentation of their configuration files and further applicable manuals. A user who is trying to understand the purpose of a unit, how it fits into the boot process and how to configure it can now easily look up this documentation with the well-known `systemctl status` command. Here's an example how this looks for `systemd-logind.service`:

```
$ systemctl status systemd-logind.service
systemd-logind.service - Login Service
   Loaded: loaded (/usr/lib/systemd/system/systemd-logind.service; static)
   Active: active (running) since Mon, 25 Jun 2012 22:39:24 +0200; 1 day and 18h ago
     Docs: man:systemd-logind.service(7)
           man:logind.conf(5)
           http://www.freedesktop.org/wiki/Software/systemd/multiseat
 Main PID: 562 (systemd-logind)
   CGroup: name=systemd:/system/systemd-logind.service
    └ 562 /usr/lib/systemd/systemd-logind

Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/
Jun 25 22:39:24 epsilon systemd-logind[562]: Watching system buttons on /dev/input/
Jun 25 22:39:25 epsilon systemd-logind[562]: New session 1 of user gdm.
Jun 25 22:39:25 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/4
Jun 25 22:39:32 epsilon systemd-logind[562]: New session 2 of user lennart.
Jun 25 22:39:32 epsilon systemd-logind[562]: Linked /tmp/.X11-unix/X0 to /run/user/5
Jun 25 22:39:54 epsilon systemd-logind[562]: Removed session 1.
```

On the first look this output changed very little. If you look closer however you will find that it now includes one new field: `Docs` lists references to the documentation of this service. In this case there are two man page URIs and one web URL specified. The man pages describe the purpose and configuration of this service, the web URL includes an introduction to the basic concepts of this service.

If the user uses a recent graphical terminal implementation it is sufficient to click on the URIs shown to get the respective documentation[2]. With other words: it never has been that easy to figure out what a specific component of our boot is about: just use `systemctl status` to get more information about it and click on the links shown to find the documentation.

The past days I have written man pages and added these references for every single unit we ship with systemd. This means, with `systemctl status` you now have a very easy way to find out more about every single service of the core OS.

If you are not using a graphical terminal (where you can just click on URIs), a man page URI in the middle of the output of `systemctl status` is not the most useful thing to have. To make reading the referenced man pages easier we have also added a new command:

```
systemctl help systemd-logind.service
```

Which will open the listed man pages right-away, without the need to click anything or copy/paste an URI.

The URIs are in the formats documented by the uri(7) man page. Units may reference http and https URLs, as well as man and info pages.

Of course all this doesn't make everything self-explanatory, simply because the user still has to find out about `systemctl status` (and even `systemctl` in the first place so that he even knows what units there are); however with this basic knowledge further help on specific units is in very easy reach.

We hope that this kind of interlinking of runtime behaviour and the matching documentation is a big step forward to make our boot easier to understand.

This functionality is partially already available in Fedora 17, and will show up in complete form in Fedora 18.

That all said, credit where credit is due: this kind of references to documentation within the service descriptions is not new, Solaris' SMF had similar functionality for quite some time. However, we believe this new systemd feature is certainly a novelty on Linux, and with systemd we now offer you the best documented and best self-explaining init system.

Of course, if you are writing unit files for your own packages, please consider also including references to the documentation of your services and its configuration. This is really easy to do, just list the URIs in the new `Documentation=` field in the `[Unit]` section of your unit files. For details see systemd.unit(5). The more comprehensively we include links to documentation in our OS services the easier the work of administrators becomes. (To make sure Fedora makes comprehensive use of this functionality I filed a bug on FPC).

Oh, and BTW: if you are looking for a rough overview of systemd's boot process here's another new man page we recently added, which includes a pretty ASCII flow chart of the boot process and the units involved.

Footnotes

[1] Which TBH is a pretty crufty, strange one on top.

[2] Well, a terminal where this bug is fixed (used together with a help browser where this one is fixed).

systemd for Administrators, Part XV

Quickly following the previous iteration, here's now the fifteenth installment of my ongoing series on systemd for Administrators:

Watchdogs

There are three big target audiences we try to cover with systemd: the embedded/mobile folks, the desktop people and the server folks. While the systems used by embedded/mobile tend to be underpowered and have few resources are available, desktops tend to be much more powerful machines -- but still much less resourceful than servers. Nonetheless there are surprisingly many features that matter to both extremes of this axis (embedded and servers), but not the center (desktops). On of them is support for watchdogs in hardware and software.

Embedded devices frequently rely on watchdog hardware that resets it automatically if software stops responding (more specifically, stops signalling the hardware in fixed intervals that it is still alive). This is required to increase reliability and make sure that regardless what happens the best is attempted to get the system working again. Functionality like this makes little sense on the desktop[1]. However, on high-availability servers watchdogs are frequently used, again.

Starting with version 183 systemd provides full support for hardware watchdogs (as exposed in `/dev/watchdog` to userspace), as well as supervisor (software) watchdog support for invidual system services. The basic idea is the following: if enabled, systemd will regularly ping the watchdog hardware. If systemd or the kernel hang this ping will not happen anymore and the hardware will automatically reset the system. This way systemd and the kernel are protected from boundless hangs -- by the hardware. To make the chain complete, systemd then exposes a software watchdog interface for individual services so that they can also be restarted (or some other action taken) if they begin to hang. This software watchdog logic can be configured individually for each service in the ping frequency and the action to take. Putting both parts together (i.e. hardware watchdogs supervising systemd and the kernel, as well as systemd supervising all other services) we have a reliable way to watchdog every single component of the system.

To make use of the hardware watchdog it is sufficient to set the `RuntimeWatchdogSec=` option in `/etc/systemd/system.conf`. It defaults to 0 (i.e. no hardware watchdog use). Set it to a value like 20s and the watchdog is enabled. After 20s of no keep-alive pings the hardware will reset itself. Note that systemd will send a ping to the hardware at half the specified interval, i.e. every 10s. And that's already all there is to it. By enabling this single, simple option you have turned on supervision by the hardware of systemd and the kernel beneath it.[2]

Note that the hardware watchdog device (`/dev/watchdog`) is single-user only. That means that you can either enable this functionality in systemd, or use a separate external watchdog daemon, such as the aptly named watchdog.

`ShutdownWatchdogSec=` is another option that can be configured in `/etc/systemd/system.conf`. It controls the watchdog interval to use during reboots. It defaults to 10min, and adds extra reliability to the system reboot logic: if a clean reboot is not possible and shutdown hangs, we rely on the watchdog hardware to reset the system abruptly, as extra safety net.

So much about the hardware watchdog logic. These two options are really everything that is necessary to make use of the hardware watchdogs. Now, let's have a look how to add watchdog logic to individual services.

First of all, to make software watchdog-supervisable it needs to be patched to send out "I am alive" signals in regular intervals in its event loop. Patching this is relatively easy. First, a daemon needs to read the `WATCHDOG_USEC=` environment variable. If it is set, it will contain the watchdog interval in usec formatted as ASCII text string, as it is configured for the service. The daemon should then issue `sd_notify("WATCHDOG=1")` calls every half of that interval. A daemon patched this way should transparently support watchdog functionality by checking whether the environment variable is set and honouring the value it is set to.

To enable the software watchdog logic for a service (which has been patched to support the logic pointed out above) it is sufficient to set the `WatchdogSec=` to the desired failure latency. See systemd.service(5) for details on this setting. This causes `WATCHDOG_USEC=` to be set for the service's processes and will cause the service to enter a failure state as soon as no keep-alive ping is received within the configured interval.

If a service enters a failure state as soon as the watchdog logic detects a hang, then this is hardly sufficient to build a reliable system. The next step is to configure whether the service shall be restarted and how often, and what to do if it then still fails. To enable automatic service restarts on failure set `Restart=on-failure` for the service. To configure how many times a service shall be attempted to be restarted use the combination of `StartLimitBurst=` and `StartLimitInterval=` which allow you to configure how often a service may restart within a time interval. If that limit is reached, a special action can be taken. This action is configured with `StartLimitAction=`. The default is a `none`, i.e. that no further action is taken and the service simply remains in the failure state without any further attempted restarts. The other three possible values are `reboot`, `reboot-force` and `reboot-immediate`. `reboot` attempts a clean reboot, going through the usual, clean shutdown logic. `reboot-force` is more abrupt: it will not actually try to cleanly shutdown any services, but immediately kills all remaining services and unmounts all file systems and then forcibly reboots (this way all file systems will be clean but reboot will still be very fast). Finally, `reboot-immediate` does not attempt to kill any process or unmount any file systems. Instead it just hard reboots the machine without delay. `reboot-immediate` hence comes closest to a reboot triggered by a hardware watchdog. All these settings are documented in systemd.service(5).

Putting this all together we now have pretty flexible options to watchdog-supervise a specific service and configure automatic restarts of the service if it hangs, plus take ultimate action if that doesn't help.

Here's an example unit file:

```
[Unit]
Description=My Little Daemon
Documentation=man:mylittled(8)

[Service]
ExecStart=/usr/bin/mylittled
WatchdogSec=30s
Restart=on-failure
StartLimitInterval=5min
```

```
StartLimitBurst=4
StartLimitAction=reboot-force
```

This service will automatically be restarted if it hasn't pinged the system manager for longer than 30s or if it fails otherwise. If it is restarted this way more often than 4 times in 5min action is taken and the system quickly rebooted, with all file systems being clean when it comes up again.

And that's already all I wanted to tell you about! With hardware watchdog support right in PID 1, as well as supervisor watchdog support for individual services we should provide everything you need for most watchdog usecases. Regardless if you are building an embedded or mobile applience, or if your are working with high-availability servers, please give this a try!

(Oh, and if you wonder why in heaven PID 1 needs to deal with `/dev/watchdog`, and why this shouldn't be kept in a separate daemon, then please read this again and try to understand that this is all about the supervisor chain we are building here, where the hardware watchdog supervises systemd, and systemd supervises the individual services. Also, we believe that a service not responding should be treated in a similar way as any other service error. Finally, pinging `/dev/watchdog` is one of the most trivial operations in the OS (basically little more than a ioctl() call), to the support for this is not more than a handful lines of code. Maintaining this externally with complex IPC between PID 1 (and the daemons) and this watchdog daemon would be drastically more complex, error-prone and resource intensive.)

Note that the built-in hardware watchdog support of systemd does not conflict with other watchdog software by default. systemd does not make use of `/dev/watchdog` by default, and you are welcome to use external watchdog daemons in conjunction with systemd, if this better suits your needs.

And one last thing: if you wonder whether your hardware has a watchdog, then the answer is: almost definitely yes -- if it is anything more recent than a few years. If you want to verify this, try the wdctl tool from recent util-linux, which shows you everything you need to know about your watchdog hardware.

I'd like to thank the great folks from Pengutronix for contributing most of the watchdog logic. Thank you!

Footnotes

[1] Though actually most desktops tend to include watchdog hardware these days too, as this is cheap to build and available in most modern PC chipsets.

[2] So, here's a free tip for you if you hack on the core OS: don't enable this feature while you hack. Otherwise your system might suddenly reboot if you are in the middle of tracing through PID 1 with gdb and cause it to be stopped for a moment, so that no hardware ping can be done…

systemd for Administrators, Part XVI

And, yes, here's now the sixteenth installment of my ongoing series on systemd for Administrators:

Gettys on Serial Consoles (and Elsewhere)

TL;DR: To make use of a serial console, just use `console=ttyS0` on the kernel command line, and systemd will automatically start a getty on it for you.

While physical RS232 serial ports have become exotic in today's PCs they play an important role in modern servers and embedded hardware. They provide a relatively robust and minimalistic way to access the console of your device, that works even when the network is hosed, or the primary UI is unresponsive. VMs frequently emulate a serial port as well.

Of course, Linux has always had good support for serial consoles, but with systemd we tried to make serial console support even simpler to use. In the following text I'll try to give an overview how serial console gettys on systemd work, and how TTYs of any kind are handled.

Let's start with the key take-away: in most cases, to get a login prompt on your serial prompt you don't need to do anything. systemd checks the kernel configuration for the selected kernel console and will simply spawn a serial getty on it. That way it is entirely sufficient to configure your kernel console properly (for example, by adding `console=ttyS0` to the kernel command line) and that's it. But let's have a look at the details:

In systemd, two template units are responsible for bringing up a login prompt on text consoles:

1. `getty@.service` is responsible for virtual terminal (VT) login prompts, i.e. those on your VGA screen as exposed in `/dev/tty1` and similar devices.
2. `serial-getty@.service` is responsible for all other terminals, including serial ports such as `/dev/ttyS0`. It differs in a couple of ways from `getty@.service`: among other things the `$TERM`

environment variable is set to `vt102` (hopefully a good default for most serial terminals) rather than `linux` (which is the right choice for VTs only), and a special logic that clears the VT scrollback buffer (and only work on VTs) is skipped.

Virtual Terminals

Let's have a closer look how `getty@.service` is started, i.e. how login prompts on the virtual terminal (i.e. non-serial TTYs) work. Traditionally, the init system on Linux machines was configured to spawn a fixed number login prompts at boot. In most cases six instances of the getty program were spawned, on the first six VTs, `tty1` to `tty6`.

In a systemd world we made this more dynamic: in order to make things more efficient login prompts are now started on demand only. As you switch to the VTs the getty service is instantiated to `getty@tty2.service`, `getty@tty5.service` and so on. Since we don't have to unconditionally start the getty processes anymore this allows us to save a bit of resources, and makes start-up a bit faster. This behaviour is mostly transparent to the user: if the user activates a VT the getty is started right-away, so that the user will hardly notice that it wasn't running all the time. If he then logs in and types `ps` he'll notice however that getty instances are only running for the VTs he so far switched to.

By default this automatic spawning is done for the VTs up to VT6 only (in order to be close to the traditional default configuration of Linux systems)[1]. Note that the auto-spawning of gettys is only attempted if no other subsystem took possession of the VTs yet. More specifically, if a user makes frequent use of fast user switching via GNOME he'll get his X sessions on the first six VTs, too, since the lowest available VT is allocated for each session.

Two VTs are handled specially by the auto-spawning logic: firstly `tty1` gets special treatment: if we boot into graphical mode the display manager takes possession of this VT. If we boot into multi-user (text) mode a getty is started on it -- unconditionally, without any on-demand logic[2].

Secondly, `tty6` is especially reserved for auto-spawned gettys and unavailable to other subsystems such as X[3]. This is done in order to ensure that there's always a way to get a text login, even if due to fast user switching X took possession of more than 5 VTs.

Serial Terminals

Handling of login prompts on serial terminals (and all other kind of non-VT terminals) is different from that of VTs. By default systemd will instantiate one `serial-getty@.service` on the main kernel[4] console, if it is not a virtual terminal. The kernel console is where the kernel outputs its own log messages and is usually configured on the kernel command line in the boot loader via an argument such as `console=ttyS0`[5]. This logic ensures that when the user asks the kernel to redirect its output onto a certain serial terminal, he will automatically also get a login prompt on it as the boot completes[6]. systemd will also spawn a login prompt on the first special VM console (that's `/dev/hvc0`, `/dev/xvc0`, `/dev/hvsi0`), if the system is run in a VM that provides these devices. This logic is implemented in a generator called systemd-getty-generator that is run early at boot and pulls in the necessary services depending on the execution environment.

In many cases, this automatic logic should already suffice to get you a login prompt when you need one, without any specific configuration of systemd. However, sometimes there's the need to manually configure a serial getty, for example, if more than one serial login prompt is needed or the kernel console should be redirected to a different terminal than the login prompt. To facilitate this it is sufficient to instantiate `serial-getty@.service` once for each serial port you want it to run on[7]:

```
# systemctl enable serial-getty@ttyS2.service
# systemctl start serial-getty@ttyS2.service
```

And that's it. This will make sure you get the login prompt on the chosen port on all subsequent boots, and starts it right-away too.

Sometimes, there's the need to configure the login prompt in even more detail. For example, if the default baud rate configured by the kernel is not correct or other `agetty` parameters need to be changed. In such a case simply copy the default unit template to `/etc/systemd/system` and edit it there:

```
# cp /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/serial-getty@
# vi /etc/systemd/system/serial-getty@ttyS2.service
 .... now make your changes to the agetty command line ...
# ln -s /etc/systemd/system/serial-getty@ttyS2.service /etc/systemd/system/getty.targ
# systemctl daemon-reload
# systemctl start serial-getty@ttyS2.service
```

This creates a unit file that is specific to serial port `ttyS2`, so that you can make specific changes to this port and this port only.

And this is pretty much all there's to say about serial ports, VTs and login prompts on them. I hope this was interesting, and please come back soon for the next installment of this series!

Footnotes

[1] You can easily modify this by changing `NAutoVTs=` in logind.conf.

[2] Note that whether the getty on VT1 is started on-demand or not hardly makes a difference, since VT1 is the default active VT anyway, so the demand is there anyway at boot.

[3] You can easily change this special reserved VT by modifying `ReserveVT=` in logind.conf.

[4] If multiple kernel consoles are used simultaneously, the *main* console is the one listed *first* in `/sys/class/tty/console/active`, which is the *last* one listed on the kernel command line.

[5] See kernel-parameters.txt for more information on this kernel command line option.

[6] Note that `agetty -s` is used here so that the baud rate configured at the kernel command line is not altered and continued to be used by the login prompt.

[7] Note that this `systemctl enable` syntax only works with systemd 188 and newer (i.e. F18). On older versions use `ln -s /usr/lib/systemd/system/serial-getty@.service /etc/systemd/system/getty.target.wants/serial-getty@ttyS2.service ; systemctl daemon-reload` instead.

systemd for Administrators, Part XVII

It's that time again, here's now the seventeenth installment of my ongoing series on systemd for Administrators:

Using the Journal

A while back I already posted a blog story introducing some functionality of the journal, and how it is exposed in `systemctl`. In this episode I want to explain a few more uses of the journal, and how you can make it work for you.

If you are wondering what the journal is, here's an explanation in a few words to get you up to speed: the journal is a component of systemd, that captures Syslog messages, Kernel log messages, initial RAM disk and early boot messages as well as messages written to STDOUT/STDERR of all services, indexes them and makes this available to the user. It can be used in parallel, or in place of a traditional syslog daemon, such as rsyslog or syslog-ng. For more information, see the initial announcement.

The journal has been part of Fedora since F17. With Fedora 18 it now has grown into a reliable, powerful tool to handle your logs. Note however, that on F17 and F18 the journal is configured by default to store logs only in a small ring-buffer in `/run/log/journal`, i.e. not persistent. This of course limits its usefulness quite drastically but is sufficient to show a bit of recent log history in `systemctl status`. For Fedora 19, we plan to change this, and enable persistent logging by default. Then, journal files will be stored in `/var/log/journal` and can grow much larger, thus making the journal a lot more useful.

Enabling Persistency

In the meantime, on F17 or F18, you can enable journald's persistent storage manually:

```
# mkdir -p /var/log/journal
```

After that, it's a good idea to reboot, to get some useful structured data into your journal to play with. Oh, and since you have the journal now, you don't need syslog anymore (unless having

/var/log/messages as text file is a necessity for you.), so you can choose to deinstall rsyslog:

```
# yum remove rsyslog
```

Basics

Now we are ready to go. The following text shows a lot of features of systemd 195 as it will be included in Fedora 18[1], so if your F17 can't do the tricks you see, please wait for F18. First, let's start with some basics. To access the logs of the journal use the journalctl(1) tool. To have a first look at the logs, just type in:

```
# journalctl
```

If you run this as root you will see all logs generated on the system, from system components the same way as for logged in users. The output you will get looks like a pixel-perfect copy of the traditional /var/log/messages format, but actually has a couple of improvements over it:

- Lines of error priority (and higher) will be highlighted red.
- Lines of notice/warning priority will be highlighted bold.
- The timestamps are converted into your local time-zone.
- The output is auto-paged with your pager of choice (defaults to less).
- This will show *all* available data, including rotated logs.
- Between the output of each boot we'll add a line clarifying that a new boot begins now.

Note that in this blog story I will not actually show you any of the output this generates, I cut that out for brevity -- and to give you a reason to try it out yourself with a current image for F18's development version with systemd 195. But I do hope you get the idea anyway.

Access Control

Browsing logs this way is already pretty nice. But requiring to be root sucks of course, even administrators tend to do most of their work as unprivileged users these days. By default, Journal users can only watch their own logs, unless they are root or in the adm group. To make watching system logs more fun, let's add ourselves to adm:

```
# usermod -a -G adm lennart
```

After logging out and back in as `lennart` I know have access to the full journal of the system and all users:

```
$ journalctl
```

Live View

If invoked without parameters journalctl will show me the current log database. Sometimes one needs to watch logs as they grow, where one previously used `tail -f /var/log/messages`:

```
$ journalctl -f
```

Yes, this does exactly what you expect it to do: it will show you the last ten logs lines and then wait for changes and show them as they take place.

Basic Filtering

When invoking `journalctl` without parameters you'll see the whole set of logs, beginning with the oldest message stored. That of course, can be a lot of data. Much more useful is just viewing the logs of the current boot:

```
$ journalctl -b
```

This will show you only the logs of the current boot, with all the aforementioned gimmicks mentioned. But sometimes even this is way too much data to process. So what about just listing all the real issues to care about: all messages of priority levels ERROR and worse, from the current boot:

```
$ journalctl -b -p err
```

If you reboot only seldom the `-b` makes little sense, filtering based on time is much more useful:

```
$ journalctl --since=yesterday
```

And there you go, all log messages from the day before at 00:00 in the morning until right now. Awesome! Of course, we can combine this with `-p err` or a similar match. But humm, we are looking for something that happened on the 15th of October, or was it the 16th?

```
$ journalctl --since=2012-10-15 --until="2011-10-16 23:59:59"
```

Yupp, there we go, we found what we were looking for. But humm, I noticed that some CGI script in Apache was acting up earlier today, let's see what Apache logged at that time:

```
$ journalctl -u httpd --since=00:00 --until=9:30
```

Oh, yeah, there we found it. But hey, wasn't there an issue with that disk `/dev/sdc`? Let's figure out what was going on there:

```
$ journalctl /dev/sdc
```

OMG, a disk error![2] Hmm, let's quickly replace the disk before we lose data. Done! Next! -- Hmm, didn't I see that the vpnc binary made a booboo? Let's check for that:

```
$ journalctl /usr/sbin/vpnc
```

Hmm, I don't get this, this seems to be some weird interaction with `dhclient`, let's see both outputs, interleaved:

```
$ journalctl /usr/sbin/vpnc /usr/sbin/dhclient
```

That did it! Found it!

Advanced Filtering

Whew! That was awesome already, but let's turn this up a notch. Internally systemd stores each log entry with a set of *implicit* meta data. This meta data looks a lot like an environment block, but actually is a bit more powerful: values can take binary, large values (though this is the exception, and usually they just contain UTF-8), and fields can have multiple values assigned (an exception too, usually they only have one value). This implicit meta data is collected for each and every log message, without user intervention. The data will be there, and wait to be used by you. Let's see how this looks:

```
$ journalctl -o verbose -n
[...]
Tue, 2012-10-23 23:51:38 CEST [s=ac9e9c423355411d87bf0ba1a9b424e8;i=4301;
        PRIORITY=6
        SYSLOG_FACILITY=3
        _MACHINE_ID=a91663387a90b89f185d4e860000001a
        _HOSTNAME=epsilon
        _TRANSPORT=syslog
        SYSLOG_IDENTIFIER=avahi-daemon
        _COMM=avahi-daemon
        _EXE=/usr/sbin/avahi-daemon
        _SYSTEMD_CGROUP=/system/avahi-daemon.service
        _SYSTEMD_UNIT=avahi-daemon.service
        _SELINUX_CONTEXT=system_u:system_r:avahi_t:s0
        _UID=70
        _GID=70
        _CMDLINE=avahi-daemon: registering [epsilon.local]
        MESSAGE=Joining mDNS multicast group on interface wlan0.IPv4 with address 172
        _BOOT_ID=5335e9cf5d954633bb99aefc0ec38c25
        _PID=27937
        SYSLOG_PID=27937
        _SOURCE_REALTIME_TIMESTAMP=1351029098747042
```

(I cut out a lot of noise here, I don't want to make this story overly long. `-n` without parameter shows you the last 10 log entries, but I cut out all but the last.)

With the `-o verbose` switch we enabled verbose output. Instead of showing a pixel-perfect copy of classic `/var/log/messages` that only includes a minimimal subset of what is available we now see all the gory details the journal has about each entry. But it's highly interesting: there is user credential information, SELinux bits, machine information and more. For a full list of common, well-known fields, see the man page.

Now, as it turns out the journal database is indexed by *all* of these fields, out-of-the-box! Let's try this out:

```
$ journalctl _UID=70
```

And there you go, this will show all log messages logged from Linux user ID 70. As it turns out one can easily combine these matches:

```
$ journalctl _UID=70 _UID=71
```

Specifying two matches for the same field will result in a logical OR combination of the matches. All entries matching either will be shown, i.e. all messages from either UID 70 or 71.

```
$ journalctl _HOSTNAME=epsilon _COMM=avahi-daemon
```

You guessed it, if you specify two matches for different field names, they will be combined with a logical AND. All entries matching both will be shown now, meaning that all messages from processes named `avahi-daemon` *and* host `epsilon`.

But of course, that's not fancy enough for us. We are computer nerds after all, we live off logical expressions. We must go deeper!

```
$ journalctl _HOSTNAME=theta _UID=70 + _HOSTNAME=epsilon _COMM=avahi-daemo
```

The + is an explicit OR you can use in addition to the implied OR when you match the same field twice. The line above hence means: show me everything from host `theta` with UID 70, or of host `epsilon` with a process name of `avahi-daemon`.

And now, it becomes magic!

That was already pretty cool, right? Righ! But heck, who can remember all those values a field can take in the journal, I mean, seriously, who has thaaaat kind of photographic memory? Well, the journal has:

```
$ journalctl -F _SYSTEMD_UNIT
```

This will show us all values the field _SYSTEMD_UNIT takes in the database, or in other words: the names of all systemd services which ever logged into the journal. This makes it super-easy to build nice matches. But wait, turns out this all is actually hooked up with shell completion on bash! This gets even more awesome: as you type your match expression you will get a list of well-known field names, and of the values they can take! Let's figure out how to filter for SELinux labels again. We remember the field name was something with SELINUX in it, let's try that:

```
$ journalctl _SE<TAB>
```

And yupp, it's immediately completed:

```
$ journalctl _SELINUX_CONTEXT=
```

Cool, but what's the label again we wanted to match for?

```
$ journalctl _SELINUX_CONTEXT=<TAB><TAB>
kernel                                     system_u:system_r:local_login_t:s0-s0:c0.c1023
system_u:system_r:accountsd_t:s0                       system_u:system_r:lvm_t:s0
system_u:system_r:avahi_t:s0                         system_u:system_r:modemmanager_
system_u:system_r:bluetooth_t:s0                       system_u:system_r:NetworkManag
system_u:system_r:chkpwd_t:s0-s0:c0.c1023                  system_u:system_r:policykit_
system_u:system_r:chronyd_t:s0                        system_u:system_r:rtkit_daemon_t:
system_u:system_r:crond_t:s0-s0:c0.c1023                   system_u:system_r:syslogd_t:
system_u:system_r:devicekit_disk_t:s0                    system_u:system_r:system_cronjo
```

```
system_u:system_r:dhcpc_t:s0                        system_u:system_r:system_dbusd_t
system_u:system_r:dnsmasq_t:s0-s0:c0.c1023             system_u:system_r:system
system_u:system_r:init_t:s0                        system_u:system_r:systemd_tmpfiles_t
```

Ah! Right! We wanted to see everything logged under PolicyKit's security label:

```
$ journalctl _SELINUX_CONTEXT=system_u:system_r:policykit_t:s0
```

Wow! That was easy! I didn't know anything related to SELinux could be thaaat easy! ;-) Of course this kind of completion works with any field, not just SELinux labels.

So much for now. There's a lot more cool stuff in journalctl(1) than this. For example, it generates JSON output for you! You can match against kernel fields! You can get simple `/var/log/messages`-like output but with *relative* timestamps! And so much more!

Anyway, in the next weeks I hope to post more stories about all the cool things the journal can do for you. This is just the beginning, stay tuned.

Footnotes

[1] systemd 195 is currently still in Bodhi but hopefully will get into F18 proper soon, and definitely before the release of Fedora 18.

[2] OK, I cheated here, indexing by block device is not in the kernel yet, but on its way due to Hannes' fantastic work, and I hope it will make appearence in F18.

systemd for Administrators, Part XVIII

Hot on the heels of the previous story, here's now the eighteenth installment of my ongoing series on systemd for Administrators:

Managing Resources

An important facet of modern computing is resource management: if you run more than one program on a single machine you want to assign the available resources to them enforcing particular policies. This is particularly crucial on smaller, embedded or mobile systems where the scarce resources are the main constraint, but equally for large installations such as cloud setups, where resources are plenty, but the number of programs/services/containers on a single node is drastically higher.

Traditionally, on Linux only one policy was really available: all processes got about the same CPU time, or IO bandwith, modulated a bit via the process *nice* value. This approach is very simple and covered the various uses for Linux quite well for a long time. However, it has drawbacks: not all all processes deserve to be even, and services involving lots of processes (think: Apache with a lot of CGI workers) this way would get more resources than services whith very few (think: syslog).

When thinking about service management for systemd, we quickly realized that resource management must be core functionality of it. In a modern world -- regardless if server or embedded -- controlling CPU, Memory, and IO resources of the various services cannot be an afterthought, but must be built-in as first-class service settings. And it must be per-service and not per-process as the traditional nice values or POSIX Resource Limits were.

In this story I want to shed some light on what you can do to enforce resource policies on systemd services. Resource Management in one way or another has been available in systemd for a while already, so it's really time we introduce this to the broader audience.

In an earlier blog post I highlighted the difference between Linux Control Croups (cgroups) as a labelled, hierarchal grouping mechanism, and Linux cgroups as a resource controlling subsystem.

While systemd requires the former, the latter is optional. And this optional latter part is now what we can make use of to manage per-service resources. (At this points, it's probably a good idea to read up on cgroups before reading on, to get at least a basic idea what they are and what they accomplish. Even thought the explanations below will be pretty high-level, it all makes a lot more sense if you grok the background a bit.)

The main Linux cgroup controllers for resource management are cpu, memory and blkio. To make use of these, they need to be enabled in the kernel, which many distributions (including Fedora) do. systemd exposes a couple of high-level service settings to make use of these controllers without requiring too much knowledge of the gory kernel details.

Managing CPU

As a nice default, if the `cpu` controller is enabled in the kernel, systemd will create a cgroup for each service when starting it. Without any further configuration this already has one nice effect: on a systemd system every system service will get an even amount of CPU, regardless how many processes it consists off. Or in other words: on your web server MySQL will get the roughly same amount of CPU as Apache, even if the latter consists a 1000 CGI script processes, but the former only of a few worker tasks. (This behavior can be turned off, see DefaultControllers= in `/etc/systemd/system.conf`.)

On top of this default, it is possible to explicitly configure the CPU shares a service gets with the CPUShares= setting. The default value is 1024, if you increase this number you'll assign more CPU to a service than an unaltered one at 1024, if you decrease it, less.

Let's see in more detail, how we can make use of this. Let's say we want to assign Apache 1500 CPU shares instead of the default of 1024. For that, let's create a new administrator service file for Apache in `/etc/systemd/system/httpd.service`, overriding the vendor supplied one in `/usr/lib/systemd/system/httpd.service`, but let's change the `CPUShares=` parameter:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
```

```
CPUShares=1500
```

The first line will pull in the vendor service file. Now, lets's reload systemd's configuration and restart Apache so that the new service file is taken into account:

```
systemctl daemon-reload
systemctl restart httpd.service
```

And yeah, that's already it, you are done!

(Note that setting `CPUShares=` in a unit file will cause the specific service to get its own cgroup in the `cpu` hierarchy, even if `cpu` is not included in `DefaultControllers=`.)

Analyzing Resource usage

Of course, changing resource assignments without actually understanding the resource usage of the services in questions is like blind flying. To help you understand the resource usage of all services, we created the tool systemd-cgtop, that will enumerate all cgroups of the system, determine their resource usage (CPU, Memory, and IO) and present them in a top-like fashion. Building on the fact that systemd services are managed in cgroups this tool hence can present to you for services what top shows you for processes.

Unfortunately, by default `cgtop` will only be able to chart CPU usage per-service for you, IO and Memory are only tracked as total for the entire machine. The reason for this is simply that by default there are no per-service cgroups in the `blkio` and `memory` controller hierarchies but that's what we need to determine the resource usage. The best way to get this data for all services is to simply add the `memory` and `blkio` controllers to the aforementioned `DefaultControllers=` setting in `system.conf`.

Managing Memory

To enforce limits on memory systemd provides the `MemoryLimit=`, and `MemorySoftLimit=` settings for services, summing up the memory of all its processes. These settings take memory sizes in bytes that are the total memory limit for the service. This setting understands the usual K, M, G, T suffixes for Kilobyte, Megabyte, Gigabyte, Terabyte (to the base of 1024).

```
.include /usr/lib/systemd/system/httpd.service

[Service]
MemoryLimit=1G
```

(Analogue to `CPUShares=` above setting this option will cause the service to get its own cgroup in the `memory` cgroup hierarchy.)

Managing Block IO

To control block IO multiple settings are available. First of all `BlockIOWeight=` may be used which assigns an IO *weight* to a specific service. In behaviour the *weight* concept is not unlike the *shares* concept of CPU resource control (see above). However, the default weight is 1000, and the valid range is from 10 to 1000:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=500
```

Optionally, per-device weights can be specified:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=/dev/disk/by-id/ata-SAMSUNG_MMCRE28G8MXP-0VBL1_DC06K01009S
```

Instead of specifiying an actual device node you also specify any path in the file system:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOWeight=/home/lennart 750
```

If the specified path does not refer to a device node systemd will determine the block device `/home/lennart` is on, and assign the bandwith weight to it.

You can even add per-device and normal lines at the same time, which will set the per-device weight for the device, and the other value as default for everything else.

Alternatively one may control explicit bandwith limits with the `BlockIOReadBandwidth=` and `BlockIOWriteBandwidth=` settings. These settings take a pair of device node and bandwith rate (in bytes per second) or of a file path and bandwith rate:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
BlockIOReadBandwith=/var/log 5M
```

This sets the maximum read bandwith on the block device backing `/var/log` to 5Mb/s.

(Analogue to `CPUShares=` and `MemoryLimit=` using any of these three settings will result in the service getting its own cgroup in the `blkio` hierarchy.)

Managing Other Resource Parameters

The options described above cover only a small subset of the available controls the various Linux control group controllers expose. We picked these and added high-level options for them since we assumed that these are the most relevant for most folks, and that they really needed a nice interface that can handle units properly and resolve block device names.

In many cases the options explained above might not be sufficient for your usecase, but a low-level kernel cgroup setting might help. It is easy to make use of these options from systemd unit files, without having them covered with a high-level setting. For example, sometimes it might be useful to set the *swappiness* of a service. The kernel makes this controllable via the `memory.swappiness` cgroup attribute, but systemd does not expose it as a high-level option. Here's how you use it nonetheless, using the low-level `ControlGroupAttribute=` setting:

```
.include /usr/lib/systemd/system/httpd.service

[Service]
ControlGroupAttribute=memory.swappiness 70
```

(Analogue to the other cases this too causes the service to be added to the memory hierarchy.)

Later on we might add more high-level controls for the various cgroup attributes. In fact, please ping us if you frequently use one and believe it deserves more focus. We'll consider adding a high-level option for it then. (Even better: send us a patch!)

Disclaimer: note that making use of the various resource controllers does have a runtime impact on the system. Enforcing resource limits comes at a price. If you do use them, certain operations do get slower. Especially the `memory` controller has (used to have?) a bad reputation to come at a performance cost.

For more details on all of this, please have a look at the documenation of the mentioned unit settings, and of the cpu, memory and blkio controllers.

And that's it for now. Of course, this blog story only focussed on the per-*service* resource settings. On top this, you can also set the more traditional, well-known per-*process* resource settings, which will then be inherited by the various subprocesses, but always only be enforced per-process. More specifically that's `IOSchedulingClass=`, `IOSchedulingPriority=`, `CPUSchedulingPolicy=`, `CPUSchedulingPriority=`, `CPUAffinity=`, `LimitCPU=` and related. These do not make use of cgroup controllers and have a much lower performance cost. We might cover those in a later article in more detail.

systemd for Administrators, Part XIX

Happy new year 2013! Here is now the nineteenth installment of my ongoing series on systemd for Administrators:

Detecting Virtualization

When we started working on systemd we had a closer look on what the various existing init scripts used on Linux where actually doing. Among other things we noticed that a number of them where checking explicitly whether they were running in a virtualized environment (i.e. in a kvm, VMWare, LXC guest or suchlike) or not. Some init scripts disabled themselves in such cases[1], others enabled themselves only in such cases[2]. Frequently, it would probably have been a better idea to check for other conditions rather than explicitly checking for virtualization, but after looking at this from all sides we came to the conclusion that in many cases explicitly conditionalizing services based on detected virtualization is a valid thing to do. As a result we added a new configuration option to systemd that can be used to conditionalize services this way: `ConditionVirtualization`; we also added a small tool that can be used in shell scripts to detect virtualization: `systemd-detect-virt(1)`; and finally, we added a minimal bus interface to query this from other applications.

Detecting whether your code is run inside a virtualized environment is actually not that hard. Depending on what precisely you want to detect it's little more than running the CPUID instruction and maybe checking a few files in `/sys` and `/proc`. The complexity is mostly about knowing the strings to look for, and keeping this list up-to-date. Currently, the the virtualization detection code in systemd can detect the following virtualization systems:

- Hardware virtualization (i.e. VMs):
  - qemu
  - kvm
  - vmware
  - microsoft
  - oracle
  - xen

    - bochs

- Same-kernel virtualization (i.e. containers):
    - chroot
    - openvz
    - lxc
    - lxc-libvirt
    - systemd-nspawn

Let's have a look how one may make use if this functionality.

Conditionalizing Units

Adding `ConditionVirtualization` to the `[Unit]` section of a unit file is enough to conditionalize it depending on which virtualization is used or whether one is used at all. Here's an example:

```
[Unit]
Name=My Foobar Service (runs only only on guests)
ConditionVirtualization=yes

[Service]
ExecStart=/usr/bin/foobard
```

Instead of specifiying "`yes`" or "`no`" it is possible to specify the ID of a specific virtualization solution (Example: "`kvm`", "`vmware`", ...), or either "`container`" or "`vm`" to check whether the kernel is virtualized or the hardware. Also, checks can be prefixed with an exclamation mark ("!") to invert a check. For further details see the manual page.

In Shell Scripts

In shell scripts it is easy to check for virtualized systems with the `systemd-detect-virt(1)` tool. Here's an example:

```
if systemd-detect-virt -q ; then
        echo "Virtualization is used:" `systemd-detect-virt`
else
        echo "No virtualization is used."
fi
```

If this tool is run it will return with an exit code of zero (success) if a virtualization solution has been found, non-zero otherwise. It will also print a short identifier of the used virtualization solution, which can be suppressed with `-q`. Also, with the `-c` and `-v` parameters it is possible to detect only kernel or only hardware virtualization environments. For further details see the manual page.

In Programs

Whether virtualization is available is also exported on the system bus:

```
$ gdbus call --system --dest org.freedesktop.systemd1 --object-path /org/freedesktop/
(<'systemd-nspawn'>,)
```

This property contains the empty string if no virtualization is detected. Note that some container environments cannot be detected directly from unprivileged code. That's why we expose this property on the bus rather than providing a library -- the bus implicitly solves the privilege problem quite nicely.

Note that all of this will only ever detect and return information about the "inner-most" virtualization solution. If you stack virtualization ("We must go deeper!") then these interfaces will expose the one the code is most directly interfacing with. Specifically that means that if a container solution is used inside of a VM, then only the container is generally detected and returned.

Footonotes

[1] For example: running certain device management service in a container environment that has no access to any physical hardware makes little sense.

[2] For example: some VM solutions work best if certain vendor-specific userspace components are running that connect the guest with the host in some way.

systemd for Administrators, Part XX

This is no time for procrastination, here is already the twentieth installment of my ongoing series on systemd for Administrators:

Socket Activated Internet Services and OS Containers

Socket Activation is an important feature of systemd. When we first announced systemd we already tried to make the point how great socket activation is for increasing parallelization and robustness of socket services, but also for simplifying the dependency logic of the boot. In this episode I'd like to explain why socket activation is an important tool for drastically improving how many services and even containers you can run on a single system with the same resource usage. Or in other words, how you can drive up the density of customer sites on a system while spending less on new hardware.

Socket Activated Internet Services

First, let's take a step back. What was *socket activation* again? -- Basically, socket activation simply means that systemd sets up listening sockets (IP or otherwise) on behalf of your services (without these running yet), and then starts (*activates*) the services as soon as the first connection comes in. Depending on the technology the services might idle for a while after having processed the connection and possible follow-up connections before they exit on their own, so that systemd will again listen on the sockets and activate the services again the next time they are connected to. For the client it is not visible whether the service it is interested in is currently running or not. The service's IP socket stays continously connectable, no connection attempt ever fails, and all connects will be processed promptly.

A setup like this lowers resource usage: as services are only running when needed they only consume resources when required. Many internet sites and services can benefit from that. For example, web site hosters will have noticed that of the multitude of web sites that are on the Internet only a tiny fraction gets a continous stream of requests: the huge majority of web sites still needs to be available all the time but gets requests only very unfrequently. With a scheme like socket activation you take benefit of this. By hosting many of these sites on a single system like this and only activating their services as necessary allows a large degree of over-commit: you can run more sites on your system than the available resources actually allow. Of course, one shouldn't over-commit too much to avoid contention during peak times.

Socket activation like this is easy to use in systemd. Many modern Internet daemons already support socket activation out of the box (and for those which don't yet it's not hard to add). Together with systemd's instantiated units support it is easy to write a pair of service and socket templates that then may be instantiated multiple times, once for each site. Then, (optionally) make use of some of the security features of systemd to nicely isolate the customer's site's services from each other (think: each customer's service should only see the home directory of the customer, everybody else's directories should be invisible), and there you go: you now have a highly scalable and reliable server system, that serves a maximum of securely sandboxed services at a minimum of resources, and all nicely done with built-in technology of your OS.

This kind of setup is already in production use in a number of companies. For example, the great folks at Pantheon are running their scalable instant Drupal system on a setup that is similar to this. (In fact, Pantheon's David Strauss pioneered this scheme. David, you rock!)

Socket Activated OS Containers

All of the above can already be done with older versions of systemd. If you use a distribution that is based on systemd, you can right-away set up a system like the one explained above. But let's take this one step further. With systemd 197 (to be included in Fedora 19), we added support for socket activating not only individual services, but *entire* OS containers. And I really have to say it at this point: this is stuff I am really excited about. ;-)

Basically, with socket activated OS containers, the host's systemd instance will listen on a number of ports on behalf of a container, for example one for SSH, one for web and one for the database, and as soon as the first connection comes in, it will spawn the container this is intended for, and pass to it all three sockets. Inside of the container, another systemd is running and will accept the sockets and then distribute them further, to the services running inside the container using normal socket activation. The SSH, web and database services will only see the inside of the container, even though they have been activated by sockets that were originally created on the host! Again, to the clients this all is not visible. That an entire OS container is spawned, triggered by simple network connection is entirely transparent to the client side.[1]

The OS containers may contain (as the name suggests) a full operating system, that might even be a different distribution than is running on the host. For example, you could run your host on Fedora, but run a number of Debian containers inside of it. The OS containers will have their own systemd init system, their own SSH instances, their own process tree, and so on, but will share a number of other facilities (such as memory management) with the host.

For now, only systemd's own trivial container manager, systemd-nspawn has been updated to support this kind of socket activation. We hope that libvirt-lxc will soon gain similar functionality. At this point, let's see in more detail how such a setup is configured in systemd using nspawn:

First, please use a tool such as `debootstrap` or yum's `--installroot` to set up a container OS tree[2]. The details of that are a bit out-of-focus for this story, there's plenty of documentation around how to do this. Of course, make sure you have systemd v197 installed inside the container. For accessing the container from the command line, consider using systemd-nspawn itself. After you configured everything properly, try to boot it up from the command line with systemd-nspawn's `-b` switch.

Assuming you now have a working container that boots up fine, let's write a service file for it, to turn the container into a systemd service on the host you can start and stop. Let's create `/etc/systemd/system/mycontainer.service` on the host:

```
[Unit]
Description=My little container

[Service]
ExecStart=/usr/bin/systemd-nspawn -jbD /srv/mycontainer 3
KillMode=process
```

This service can already be started and stopped via `systemctl start` and `systemctl stop`. However, there's no nice way to actually get a shell prompt inside the container. So let's add SSH to it, and even more: let's configure SSH so that a connection to the container's SSH port will socket-activate the entire container. First, let's begin with telling the host that it shall now listen on the SSH port of the container. Let's create `/etc/systemd/system/mycontainer.socket` on the host:

```
[Unit]
Description=The SSH socket of my little container

[Socket]
ListenStream=23
```

If we start this unit with `systemctl start` on the host then it will listen on port 23, and as soon as a connection comes in it will activate our container service we defined above. We pick port 23 here, instead of the usual 22, as our host's SSH is already listening on that. nspawn virtualizes the process list and the file system tree, but does not actually virtualize the network stack, hence we just pick different ports for the host and the various containers here.

Of course, the system inside the container doesn't yet know what to do with the socket it gets passed due to socket activation. If you'd now try to connect to the port, the container would start-up but the incoming connection would be immediately closed since the container can't handle it yet. Let's fix that!

All that's necessary for that is teach SSH inside the container socket activation. For that let's simply write a pair of socket and service units for SSH. Let's create `/etc/systemd/system/sshd.socket` in the container:

```
[Unit]
Description=SSH Socket for Per-Connection Servers

[Socket]
ListenStream=23
Accept=yes
```

Then, let's add the matching SSH service file `/etc/systemd/system/sshd@.service` in the container:

```
[Unit]
Description=SSH Per-Connection Server for %I

[Service]
ExecStart=-/usr/sbin/sshd -i
StandardInput=socket
```

Then, make sure to hook `sshd.socket` into the `sockets.target` so that unit is started automatically when the container boots up:

```
ln -s /etc/systemd/system/sshd.socket /etc/systemd/system/sockets.target.wants/
```

And that's it. If we now activate `mycontainer.socket` on the host, the host's systemd will bind the socket and we can connect to it. If we do this, the host's systemd will activate the container, and pass the socket in to it. The container's systemd will then take the socket, match it up with `sshd.socket` inside the container. As there's still our incoming connection queued on it, it will then immediately trigger an instance of `sshd@.service`, and we'll have our login.

And that's already everything there is to it. You can easily add additional sockets to listen on to `mycontainer.socket`. Everything listed therein will be passed to the container on activation, and will be matched up as good as possible with all socket units configured inside the container. Sockets that cannot be matched up will be closed, and sockets that aren't passed in but are configured for listening will be bound be the container's systemd instance.

So, let's take a step back again. What did we gain through all of this? Well, basically, we can now offer a number of full OS containers on a single host, and the containers can offer their services without running continously. The density of OS containers on the host can hence be increased drastically.

Of course, this only works for kernel-based virtualization, not for hardware virtualization. i.e. something like this can only be implemented on systems such as libvirt-lxc or nspawn, but not in qemu/kvm.

If you have a number of containers set up like this, here's one cool thing the journal allows you to do. If you pass `-m` to `journalctl` on the host, it will automatically discover the journals of all local containers and interleave them on display. Nifty, eh?

With systemd 197 you have everything to set up your own socket activated OS containers on-board. However, there are a couple of improvements we're likely to add soon: for example, right now even if all services inside the container exit on idle, the container still will stay around, and we really should make it exit on idle too, if all its services exited and no logins are around. As it turns out we already have much of the infrastructure for this around: we can reuse the auto-suspend functionality we added for laptops: detecting when a laptop is idle and suspending it then is a very similar problem to detecting when a container is idle and shutting it down then.

Anyway, this blog story is already way too long. I hope I haven't lost you half-way already with all this talk of virtualization, sockets, services, different OSes and stuff. I hope this blog story is a good starting point for setting up powerful highly scalable server systems. If you want to know more, consult the documentation and drop by our IRC channel. Thank you!

Footnotes

[1] And BTW, this is another reason why fast boot times the way systemd offers them are actually a really good thing on servers, too.

[2] To make it easy: you need a command line such as `yum --releasever=19 --nogpg --installroot=/srv/mycontainer/ --disablerepo='*' --enablerepo=fedora install systemd passwd yum fedora-release vim-minimal` to install Fedora, and `debootstrap --arch=amd64 unstable /srv/mycontainer/` to install Debian. Also see the bottom of systemd-nspawn(1). Also note that auditing is currently broken for containers, and if enabled in the kernel will cause all kinds of errors in the container. Use `audit=0` on the host's kernel command line to turn it off.

systemd For Administrators, Part XXI

Container Integration

Since a while containers have been one of the hot topics on Linux. Container managers such as libvirt-lxc, LXC or Docker are widely known and used these days. In this blog story I want to shed some light on systemd's integration points with container managers, to allow seamless management of services across container boundaries.

We'll focus on OS containers here, i.e. the case where an init system runs inside the container, and the container hence in most ways appears like an independent system of its own. Much of what I describe here is available on pretty much any container manager that implements the logic described here, including libvirt-lxc. However, to make things easy we'll focus on systemd-nspawn, the mini-container manager that is shipped with systemd itself. systemd-nspawn uses the same kernel interfaces as the other container managers, however is less flexible as it is designed to be a container manager that is as simple to use as possible and "just works", rather than trying to be a generic tool you can configure in every low-level detail. We use systemd-nspawn extensively when developing systemd.

Anyway, so let's get started with our run-through. Let's start by creating a Fedora container tree in a subdirectory:

```
# yum -y --releasever=20 --nogpg --installroot=/srv/mycontainer --disablerepo='*' --en
```

This downloads a minimal Fedora system and installs it in in `/srv/mycontainer`. This command line is Fedora-specific, but most distributions provide similar functionality in one way or another. The examples section in the systemd-nspawn(1) man page contains a list of the various command lines for other distribution.

We now have the new container installed, let's set an initial root password:

```
# systemd-nspawn -D /srv/mycontainer
Spawning container mycontainer on /srv/mycontainer
Press ^] three times within 1s to kill container.
```

```
-bash-4.2# passwd
Changing password for user root.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
-bash-4.2# ^D
Container mycontainer exited successfully.
#
```

We use systemd-nspawn here to get a shell in the container, and then use passwd to set the root password. After that the initial setup is done, hence let's boot it up and log in as root with our new password:

```
$ systemd-nspawn -D /srv/mycontainer -b
Spawning container mycontainer on /srv/mycontainer.
Press ^] three times within 1s to kill container.
systemd 208 running in system mode. (+PAM +LIBWRAP +AUDIT +SELINUX +IMA +SY
Detected virtualization 'systemd-nspawn'.

Welcome to Fedora 20 (Heisenbug)!

[  OK  ] Reached target Remote File Systems.
[  OK  ] Created slice Root Slice.
[  OK  ] Created slice User and Session Slice.
[  OK  ] Created slice System Slice.
[  OK  ] Created slice system-getty.slice.
[  OK  ] Reached target Slices.
[  OK  ] Listening on Delayed Shutdown Socket.
[  OK  ] Listening on /dev/initctl Compatibility Named Pipe.
[  OK  ] Listening on Journal Socket.
         Starting Journal Service...
[  OK  ] Started Journal Service.
[  OK  ] Reached target Paths.
         Mounting Debug File System...
         Mounting Configuration File System...
         Mounting FUSE Control File System...
         Starting Create static device nodes in /dev...
         Mounting POSIX Message Queue File System...
         Mounting Huge Pages File System...
[  OK  ] Reached target Encrypted Volumes.
[  OK  ] Reached target Swap.
         Mounting Temporary Directory...
         Starting Load/Save Random Seed...
[  OK  ] Mounted Configuration File System.
[  OK  ] Mounted FUSE Control File System.
[  OK  ] Mounted Temporary Directory.
[  OK  ] Mounted POSIX Message Queue File System.
[  OK  ] Mounted Debug File System.
[  OK  ] Mounted Huge Pages File System.
[  OK  ] Started Load/Save Random Seed.
[  OK  ] Started Create static device nodes in /dev.
[  OK  ] Reached target Local File Systems (Pre).
[  OK  ] Reached target Local File Systems.
         Starting Trigger Flushing of Journal to Persistent Storage...
         Starting Recreate Volatile Files and Directories...
[  OK  ] Started Recreate Volatile Files and Directories.
         Starting Update UTMP about System Reboot/Shutdown...
[  OK  ] Started Trigger Flushing of Journal to Persistent Storage.
[  OK  ] Started Update UTMP about System Reboot/Shutdown.
[  OK  ] Reached target System Initialization.
[  OK  ] Reached target Timers.
[  OK  ] Listening on D-Bus System Message Bus Socket.
[  OK  ] Reached target Sockets.
[  OK  ] Reached target Basic System.
         Starting Login Service...
         Starting Permit User Sessions...
         Starting D-Bus System Message Bus...
[  OK  ] Started D-Bus System Message Bus.
         Starting Cleanup of Temporary Directories...
[  OK  ] Started Cleanup of Temporary Directories.
[  OK  ] Started Permit User Sessions.
         Starting Console Getty...
[  OK  ] Started Console Getty.
[  OK  ] Reached target Login Prompts.
[  OK  ] Started Login Service.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.

Fedora release 20 (Heisenbug)
Kernel 3.18.0-0.rc4.git0.1.fc22.x86_64 on an x86_64 (console)

mycontainer login: root
Password:
-bash-4.2#
```

Now we have everything ready to play around with the container integration of systemd. Let's have a look at the first tool, machinectl. When run without parameters it shows a list of all locally running containers:

```
$ machinectl
MACHINE                          CONTAINER SERVICE
mycontainer                      container nspawn

1 machines listed.
```

The "status" subcommand shows details about the container:

```
$ machinectl status mycontainer
mycontainer:
       Since: Mi 2014-11-12 16:47:19 CET; 51s ago
      Leader: 5374 (systemd)
     Service: nspawn; class container
        Root: /srv/mycontainer
     Address: 192.168.178.38
              10.36.6.162
              fd00::523f:56ff:fe00:4994
              fe80::523f:56ff:fe00:4994
          OS: Fedora 20 (Heisenbug)
        Unit: machine-mycontainer.scope
              ├─5374 /usr/lib/systemd/systemd
              └─system.slice
                ├─dbus.service
                │ └─5414 /bin/dbus-daemon --system --address=systemd: --nofork --nopidfile
                ├─systemd-journald.service
                │ └─5383 /usr/lib/systemd/systemd-journald
                ├─systemd-logind.service
                │ └─5411 /usr/lib/systemd/systemd-logind
                └─console-getty.service
                  └─5416 /sbin/agetty --noclear -s console 115200 38400 9600
```

With this we see some interesting information about the container, including its control group tree (with processes), IP addresses and root directory.

The "login" subcommand gets us a new login shell in the container:

```
# machinectl login mycontainer
Connected to container mycontainer. Press ^] three times within 1s to exit session.

Fedora release 20 (Heisenbug)
Kernel 3.18.0-0.rc4.git0.1.fc22.x86_64 on an x86_64 (pts/0)

mycontainer login:
```

The "reboot" subcommand reboots the container:

```
# machinectl reboot mycontainer
```

The "poweroff" subcommand powers the container off:

```
# machinectl poweroff mycontainer
```

So much about the machinectl tool. The tool knows a couple of more commands, please check the man page for details. Note again that even though we use systemd-nspawn as container manager here the concepts apply to any container manager that implements the logic described here, including libvirt-lxc for example.

machinectl is not the only tool that is useful in conjunction with containers. Many of systemd's own tools have been updated to explicitly support containers too! Let's try this (after starting the container up again first, repeating the systemd-nspawn command from above.):

```
# hostnamectl -M mycontainer set-hostname "wuff"
```

This uses hostnamectl(1) on the local container and sets its hostname.

Similar, many other tools have been updated for connecting to local containers. Here's systemctl(1)'s `-M` switch in action:

```
# systemctl -M mycontainer
UNIT                                 LOAD   ACTIVE SUB       DESCRIPTION
-.mount                              loaded active mounted   /
dev-hugepages.mount                  loaded active mounted   Huge Pages File System
dev-mqueue.mount                     loaded active mounted   POSIX Message Queue File Sy
proc-sys-kernel-random-boot_id.mount loaded active mounted   /proc/sys/kernel/rand
[...]
time-sync.target                     loaded active active    System Time Synchronized
timers.target                        loaded active active    Timers
systemd-tmpfiles-clean.timer         loaded active waiting   Daily Cleanup of Temporary Di
```

```
LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.

49 loaded units listed. Pass --all to see loaded but inactive units, too.
To show all installed unit files use 'systemctl list-unit-files'.
```

As expected, this shows the list of active units on the specified container, not the host. (Output is shortened here, the blog story is already getting too long).

Let's use this to restart a service within our container:

```
# systemctl -M mycontainer restart systemd-resolved.service
```

systemctl has more container support though than just the `-M` switch. With the `-r` switch it shows the units running on the host, plus all units of all local, running containers:

```
# systemctl -r
UNIT                                 LOAD   ACTIVE SUB       DESCRIPTION
boot.automount                               loaded active waiting   EFI System Partition Automoun
proc-sys-fs-binfmt_misc.automount            loaded active waiting   Arbitrary Executable Fil
sys-devices-pci0000:00-0000:00:02.0-drm-card0-card0\x2dLVDS\x2d1-intel_backlight.
[...]
timers.target                                                                   loaded active active    T
mandb.timer                                                                      loaded active waiting
systemd-tmpfiles-clean.timer                                                          loaded active wa
mycontainer:-.mount                                                                 loaded active mou
mycontainer:dev-hugepages.mount                                                             loaded act
mycontainer:dev-mqueue.mount                                                              loaded activ
[...]
mycontainer:time-sync.target                                                           loaded active ac
mycontainer:timers.target                                                            loaded active acti
mycontainer:systemd-tmpfiles-clean.timer                                                      loaded ac

LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.

191 loaded units listed. Pass --all to see loaded but inactive units, too.
To show all installed unit files use 'systemctl list-unit-files'.
```

We can see here first the units of the host, then followed by the units of the one container we have currently running. The units of the containers are prefixed with the container name, and a colon (":"). (The output is shortened again for brevity's sake.)

The `list-machines` subcommand of systemctl shows a list of all running containers, inquiring the system managers within the containers about system state and health. More specifically it shows if containers are properly booted up, or if there are any failed services:

```
# systemctl list-machines
NAME         STATE   FAILED JOBS
delta (host) running      0    0
mycontainer  running      0    0
miau         degraded     1    0
waldi        running      0    0

4 machines listed.
```

To make things more interesting we have started two more containers in parallel. One of them has a failed service, which results in the machine state to be `degraded`.

Let's have a look at journalctl(1)'s container support. It too supports `-M` to show the logs of a specific container:

```
# journalctl -M mycontainer -n 8
Nov 12 16:51:13 wuff systemd[1]: Starting Graphical Interface.
Nov 12 16:51:13 wuff systemd[1]: Reached target Graphical Interface.
Nov 12 16:51:13 wuff systemd[1]: Starting Update UTMP about System Runlevel Chang
Nov 12 16:51:13 wuff systemd[1]: Started Stop Read-Ahead Data Collection 10s After C
Nov 12 16:51:13 wuff systemd[1]: Started Update UTMP about System Runlevel Chang
Nov 12 16:51:13 wuff systemd[1]: Startup finished in 399ms.
Nov 12 16:51:13 wuff sshd[35]: Server listening on 0.0.0.0 port 24.
Nov 12 16:51:13 wuff sshd[35]: Server listening on :: port 24.
```

However, it also supports `-m` to show the combined log stream of the host and all local containers:

```
# journalctl -m -e
```

(Let's skip the output here completely, I figure you can extrapolate how this looks.)

But it's not only systemd's own tools that understand container support these days, procps sports support for it, too:

```
# ps -eo pid,machine,args
 PID MACHINE                         COMMAND
   1 -                               /usr/lib/systemd/systemd --switched-root --system --deserialize 2
[...]
2915 -                               emacs contents/projects/containers.md
3403 -                               [kworker/u16:7]
3415 -                               [kworker/u16:9]
4501 -                               /usr/libexec/nm-vpnc-service
4519 -                               /usr/sbin/vpnc --non-inter --no-detach --pid-file /var/run/Netwo
4749 -                               /usr/libexec/dconf-service
4980 -                               /usr/lib/systemd/systemd-resolved
5006 -                               /usr/lib64/firefox/firefox
5168 -                               [kworker/u16:0]
5192 -                               [kworker/u16:4]
5193 -                               [kworker/u16:5]
5497 -                               [kworker/u16:1]
5591 -                               [kworker/u16:8]
5711 -                               sudo -s
5715 -                               /bin/bash
```

```
5749 -                  /home/lennart/projects/systemd/systemd-nspawn -D /srv/myc
5750 mycontainer              /usr/lib/systemd/systemd
5799 mycontainer              /usr/lib/systemd/systemd-journald
5862 mycontainer              /usr/lib/systemd/systemd-logind
5863 mycontainer              /bin/dbus-daemon --system --address=systemd: --nofor
5868 mycontainer              /sbin/agetty --noclear --keep-baud console 115200 384
5871 mycontainer              /usr/sbin/sshd -D
6527 mycontainer              /usr/lib/systemd/systemd-resolved
[...]
```

This shows a process list (shortened). The second column shows the container a process belongs to. All processes shown with "-" belong to the host itself.

But it doesn't stop there. The new "sd-bus" D-Bus client library we have been preparing in the systemd/kdbus context knows containers too. While you use `sd_bus_open_system()` to connect to your local host's system bus `sd_bus_open_system_container()` may be used to connect to the system bus of any local container, so that you can execute bus methods on it.

`sd-login.h` and machined's bus interface provide a number of APIs to add container support to other programs too. They support enumeration of containers as well as retrieving the machine name from a PID and similar.

systemd-networkd also has support for containers. When run inside a container it will by default run a DHCP client and IPv4LL on any veth network interface named `host0` (this interface is special under the logic described here). When run on the host networkd will by default provide a DHCP server and IPv4LL on veth network interface named `ve-` followed by a container name.

Let's have a look at one last facet of systemd's container integration: the hook-up with the name service switch. Recent systemd versions contain a new NSS module nss-mymachines that make the names of all local containers resolvable via `gethostbyname()` and `getaddrinfo()`. This only applies to containers that run within their own network namespace. With the systemd-nspawn command shown above the the container shares the network configuration with the host however; hence let's restart the container, this time with a virtual `veth` network link between host and container:

```
# machinectl poweroff mycontainer
# systemd-nspawn -D /srv/mycontainer --network-veth -b
```

Now, (assuming that networkd is used in the container and outside) we can already ping the container using its name, due to the simple magic of nss-mymachines:

```
# ping mycontainer
PING mycontainer (10.0.0.2) 56(84) bytes of data.
64 bytes from mycontainer (10.0.0.2): icmp_seq=1 ttl=64 time=0.124 ms
64 bytes from mycontainer (10.0.0.2): icmp_seq=2 ttl=64 time=0.078 ms
```

Of course, name resolution not only works with `ping`, it works with all other tools that use libc `gethostbyname()` or `getaddrinfo()` too, among them venerable `ssh`.

And this is pretty much all I want to cover for now. We briefly touched a variety of integration points, and there's a lot more still if you look closely. We are working on even more container integration all the time, so expect more new features in this area with every systemd release.

Note that the whole *machine* concept is actually not limited to containers, but covers VMs too to a certain degree. However, the integration is not as close, as access to a VM's internals is not as easy as for containers, as it usually requires a network transport instead of allowing direct syscall access.

Anyway, I hope this is useful. For further details, please have a look at the linked man pages and other documentation.