Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Building Murphy-compatible embedded Linux systems

Gilad Ben-Yossef
Codefidence Ltd.
`gilad@codefidence.com`

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

— *Murphy's Law of Technology #5* [Murphy]

Abstract

It's 2:00 a.m. An embedded Linux system in the ladies' room of an Albuquerque gas station is being updated remotely. Just as the last bytes hit the flash, disaster strikes—the power fails. Now what? The regular way of updating the configuration or performing software upgrade of Linux systems is a *nonsequitur* in the embedded space. Still, many developers use these methods, or worse, for lack of a better alternative. This talk introduces a better alternative—a framework for safe remote configuration and software upgrade of a Linux system that supports atomic transactions, parallel, interactive and programmed updates, and multiple software versions with rollback and all using using such "novel" concepts as POSIX `rename(2)`, Linux `pivot_root(2)`, and the initrd/initramfs mechanism.

1 Introduction: When bad things happen to good machines

Building embedded systems, Linux-based or otherwise, involves a lot of effort. Thought must be given to designing important aspects of the system as its performance, real time constraints, hardware interfaces, and cost.

All too often, the issue of system survivability in face of Murphy's Law is not addressed as part of the overall design. Alternatively, it may be delegated to the implementor of specific parts of the overall system as "implementation details."

To understand what we mean by "system survivability in face of Murphy's law," let us consider the common warning often encountered when one updates the firmware of an embedded system:

> *"Whatever happens, DO NOT pull the plug or reboot this system until the firmware update has been completed or you risk turning this system into a brick."*

If there is something we can guarantee with certainty, while reading such a sincere warning, it is that somewhere and some when the power will indeed falter or the machine reboot just as those last precious bits are written to flash, rendering the system completely unusable.

It is important to note that this eventuality, although sure to happen, is not common. Indeed, the system can undergo thousands of firmware upgrades in the QA lab without an incident; there seems to be some magical quality to the confinements of QA labs that stops this sort of thing from happening.

Indeed, any upgrade of a a non-critical piece of equipment in an idle Tuesday afternoon is considered quite safe in the eyes of the authors, with relation to the phenomena that we are discussing.

However, any critical system upgrade, performed on a late Friday afternoon is almost guaranteed to trigger a complex chain of events involving power failures, stray cats, or the odd meteorite or two, all leading to the some sad (yet expected) outcome—a $3k or $50 irreplaceable brick.

In essence therefore, system survivability in face of Murphy's Law is defined as the chances of a given system to function in face of failure in the "worst possible time."

Despite the humorous tone chosen above, this characteristic of embedded system has a very serious and direct consequence on the bottom line: a 0.1% RMA[1] rate for a wireless router device, or a single melt down of a critical core router in a strategic customer site can spell the difference between a successful project or a failed one. Despite this, all too often design requirements and QA processes do not take Murphy's Law into account, leading to a serious issue which is only detected in the most painful way by a customer, after the product has been shipped.

[1]Return Materials Authorization, frequently used to refer to all returned product, whether authorized or not.

If there is a way therefore, to build Murphy-compliant systems, as it were, that will survive the worse possible scenario without costing the implementor too much money or time, it will be a great boon to society, not to mention embedded system developers.

As always, a trade off is at work here: for example, we can coat the system developed with a thick layer of lead, thus protecting it from damage by cosmic rays. This, however, is not very logical to do—the price-to-added-protection ratio is simply not attractive enough.

We must therefore pick our battles wisely.

In the course of a 7-year career working on building GNU/Linux-based embedded systems, we have identified two points of failure which we believe based on anecdotal evidence to be responsible for a significant number of embedded system failures, and that are easily addressable with no more then a little premeditative thought and the GNU/Linux feature set. In this paper we describe those points and suggest an efficient way to address them when developing GNU/Linux-based embedded systems. Those points are:

- Embedded system configuration
- Embedded system software upgrade

The lessons we talk about were learned the hard way: three different products sold in the market today (by Juniper Networks Inc., Finjan Software Inc., and BeyondSecurity Ltd.) already make use of ideas or whole parts of the system we're about to introduce here, and more products are on the way. In addition, as we will later disclose—we are not the first going down this road, but more on that later.

The rest of this paper is outlined as follows: In Section 2 we present current approaches, their weaknesses and strengths. In Section 3 we present the requirements from a system which will have all the strengths of the current approaches but none of their weaknesses. In Section 4 we present our approach to solving the problem of embedded system configuration: `cfgsh`, the configuration shell. In Section 5 we present our approach to solving the problem of embedded system upgrade: `sysup`: the system upgrade utility. In Section 6 we discuss future directions, and we conclude in Section 7.

2 Current approaches: The good, the bad, and the ugly

In this section we will present two of the more common approaches: the "naïve" approach and the RTOS approach. We will discuss each approach as to its merits and faults.

2.1 The "naïve" approach: tar balls and rc files

When a developer familiar with the Unix way is faced with the task of building a GNU/Linux-based embedded system, his or her tendency when it comes to handling configuration files and software update is to mimic the way such tasks are traditionally handled in Unix based workstation or servers [Embedded Linux Systems]. The flash device is used in the same way a hard disk is used in a traditional GNU/Linux workstation or server.

System configuration state, such as IP addresses, host name, or the like, is stored in small text files which are read by scripts being run by the `init(8)` process at system startup. Updating the configuration calls for editing the text files and possibly re-running the scripts.

In a similar fashion, a software upgrade is done by downloading and opening tar files of binaries which replace the system binaries and restarting the relevant processes. The more "advanced" developers forgo tar files in favor of plain cpio archives, RPM, deb files, ipkg or proprietary formats which are essentially file archives as well.

2.1.1 The good: the Unix way

The strengths of this approach are self evident: this approach makes use of the Unix "Everything is a file" paradigm, configuration files are written in the universal interface of plain text, and since the system behaves like a regular GNU/Linux workstation or server installation, it's easy to build and debug.

In addition, because all the components of a software version are just files in a file system, one can replace individual files during system operation, offering an easy "patch" facility. In the development and QA labs, this is a helpful feature.

2.1.2 The bad: no atomic transactions

A power loss during a configuration or software update may result in a system at an inconsistent state. Since the operations being performed in either case are non atomic replacements of files, a power loss in the middle of a configuration change or a system upgrade can leave some of the files in a pre-changed status while the rest of the files have already been updated and the system is no longer in a consistent state.

Inconsistent here really can mean anything at all: from a system that boots with the wrong IP, through a system which behaves strangely or fails in various ways due to incompatible library versions, and all the way up to a system that will not boot at all.

Considering that many embedded devices are being upgraded and managed via a network, a system with the wrong (or no) IP address may be as useless as a system which does not boot, when you are on the wrong side of the continent or even on a different continent altogether.

In addition, the ease of replacing single files, which is considered a boon in the development and QA labs, is a software-versions nightmare at the customer site. The ability to patch single files at a customer site gives rise to a multitude of unofficial mini-versions of the software. Thus, when a bug report comes in, how can one tell if the software really is "version 1.6" as the report says and not "version 1.6 with that patch we sent to this one customer to debug the problem but that the guys from professional services decided to put on each installation since"? The sad answer is: you can't.

2.1.3 The ugly: user interface

Editing configuration files and scripts or opening tar files is not an acceptable interface for the user of a embedded device. A tool has to be written to supply a decent interface for the user.

Given the lack of any such standard tool, every GNU/Linux-based embedded system developer seems to write one of its own. Sometimes, when there is a need for a configuration solution that spans telnet and serial CLI, web,and SNMP interfaces, three different configuration tools are written.

2.2 The RTOS approach: what we did in that other project

The RTOS[2] approach is favored by people experienced with legacy RTOS systems, which seldom have a file system at their disposal, because it costs extra.

The basic idea is that both configuration information and software versions are kept as blobs of data directly on the system flash.

Configuration is changed by mapping the flash disk memory and modifying the configuration parameters in place.

Software update is performed by overwriting whole images of the system software, comprised of the kernel, initrd or initramfs images to the flash. Some developers utilize an Ext2 [ext2] ram disk image, which leaves the system running from a read/write, but volatile environment.

Other developers prefer to use Cramfs [cramfs] or Squashfs [Squashfs] file systems, in which the root file system is read-only.

2.2.1 The good

The RTOS approach enjoys two advantages: atomic system upgrade (under certain conditions) and manageability of system software versions while possible retaining the ability to update single files at the lab.

Because system software is a single blob of data, we can achieve a sort of atomic update ability by having two separate partitions to store two versions of the system software. In

[2]For marketing reasons, most embedded OS vendors call their offering a Real Time OS, even if most of the projects using them have negligible real time requirements, if any.

this scenario, software update is performed by writing the new version firmware to the partition we *didn't* boot from, verify the write and marking in the configuration part of the flash the partition we just wrote to as the active one and booting from it.

In addition, because software updates are performed on entire file systems images, we need not worry about the software version nightmare stemming from the ability to update single files as we described in the Section 2.1.2 previously.

Furthermore, if we happen to be using a read/write but volatile root file system (such as a ram disk), we allow the developer the freedom to patch single files at run time, while having the safety guard of having all these changes rolled back automatically in the next reboot.

2.2.2 The bad

However, utilizing this advanced method requires additional flash space and a customized boot loader that can switch boot partition based on configuration information stored on the flash. Even then, we are required to prepare in advance a partition for each possible software version, which in practice leads only supporting two versions at a time.

In addition, booting into a verified firmware version with a major bug might still turn the machine into a brick.

As for the configuration information—it is kept as binary data on a flash, which is an inflexible and unforgiving format, hard to debug, and hard to backup.

2.2.3 The ugly

This approach suffers from the same need for a user interface as the naïve approach. While the approach based on standard Unix configuration files can at least rely on some common infrastructure to read and update its files, the RTOS approach dictates the creation of a proprietary tool to read the binary format in which the configuration is written on the flash.

Moreover, if different user interfaces are required to handle the configuration of the system (for example: telnet and serial CLI, web and SNMP interfaces) three different tools will have to be written or at least some common library that allows all three interfaces to cooperate in managing the configuration.

3 Requirements: building a better solution

In this section we present the requirements from a solution for updating and configuring an embedded system. These requirements are derived from the merits of existing approaches, while leaving out the limitations.

The selected approach should follow the following guidelines:

1. Allow atomic update of configuration and software versions.
2. Not require any special boot loader software.
3. Allow an update of individual files of the system software, but in a controlled fashion.
4. Everything that can be represented as a file, should.
5. Configuration files should be human readable and editable.

6. Offer a standard unified tools to deal with configuration and version management.

As we have seen, the naïve approach follows guidelines 2, 4, and 5 but fails to meet guidelines 1, 3, and 6. On the other hand the RTOS approach follows guidelines 1 and 3, although both of them optionally, and fails to meet guidelines 2, 4, 5, and 6.

It should be pointed out that both the approaches we introduced are only examples. One can think of many other approaches that follow some of the 6 guidelines but not all of them. Looking at the two approaches described above we can understand why—choosing one or the other of them is a trade off: it mandates choosing which of the guidelines you are willing to give up for the others.

Another thing worth mentioning is that there is no tool currently known to the authors which will be a good candidate to satisfy guideline 6. This is surprising, since the embedded GNU/Linux field is not short of such embedded space infrastructure (or framework): the busybox meta-utility maintained by Eric Anderson and friends or the crosstool script by Dan Kegel are two prime examples of such software which most (if not all) embedded GNU/Linux systems are built upon[3].

Still, no common framework exists today that deals with configuration and software upgrade of embedded systems in the same way that Busybox deals with system utilities and crosstool with building cross tool chains and which allows the embedded developer to build upon to create his or her respective systems.

[3]And which the authors of this article will gladly sacrifice a goat or two in order to show their gratitude to their maintainers if not for the very real fear of scaring them off from doing any additional work on their respective tools…

Can there really exist a solution which will allow us to follow all 6 guidelines with no compromises or do embedded systems are too tied up to their unique hardware platforms to give rise to such a unified tool? And if such a tool is made, will it need to be a complex and costly-to-implement solution requiring changes in the kernel, or a simple straightforward solution requiring no more than some knowledge in C?

Since you're reading this paper, you're probably assuming that we did came up with something in the end and you're perfectly right. But before we are going to tell you all about it we need to get something off of our chest first: we didn't really invent this solution at all.

Rather, when faced with the daunting task of building the perfect embedded configuration and upgrade tool(s) we chose to "stand on the shoulders of giants" and simply went off and found the best example we could lay our hands on and imitated it.

Our victim was the Cisco family of routers and its IOS operating system. Since we have observed that this specific product of embedded devices does seem to follow all of these guidelines, we naturally asked ourselves, "How did they do that?"

Cisco embedded products, however, do not run on GNU/Linux, our embedded OS of choice, nor does Cisco shares the code to its OS with the world [4]. What we are about to describe in the next chapters is therefore, how to get the same useful feature set of the Cisco line of embedded devices when using GNU/Linux—all implemented as Free Software.

[4]At least not willingly...

4 cfgsh – an embedded GNU / Linux configuration shell

cfgsh is an embedded GNU/Linux system configuration shell. It is a small C utility which aims to provides a unified standard way of handling the configuration of a GNU/Linux-based embedded system.

cfgsh was independently implemented from scratch, though it is influenced by the Cisco IOS shell. cfgsh supports three modes: an interactive mode, a setup mode, and a silent mode. Those modes will be described in the following subsections.

4.1 Interactive mode

Interactive mode gives a user an easy text-based user interface to manage the configuration, complete with menus, context sensitive help and command line completion. This is the default mode.

Upon entering the program, the user is presented with a prompt of the host name of the machine. The user can then manage the system configuration by entering commands. On-line help is available for all menus.

The GNU readline library [GNU Readline] is used to implement all the interaction with the user.

Figure 4.1 shows cfgsh main help menu.

The user may enter a sub-menu by entering the proper command. Upon doing so, the prompt changes to reflect the menu level the user is at that moment.

Figure 2 shows how the network menu is entered.

```
linbox>help
    role   Display or set system role: role
           [role].
timezone   Display or set time zone: timezone
           [time zone].
 network   Enter network configuration mode:
           network.
    ping   Ping destination: ping <hostname |
           address>.
hostname   Displays or set the host name: host-
           name [name].
    halt   Shutdown.
  reboot   Reboot.
    show   Display settings: show [config | in-
           terfaces | routes | resolver].
    save   Save configuration.
    exit   Logout.
    quit   Logout.
    help   Display this text.
linbox>
```

Figure 1: cfgsh main menu help

At any stage the user may utilize the online context-sensitive line help by simply pressing the [TAB] key. If the user is entering a command, the result is simple command completion. If the user has already specified a command and she is requesting help with the parameters, she will get either a short help text on the command parameters or parameter completion, where appropriate.

Figure 3 shows the command-line completion for the “timezone” command[5]

Every change of configuration requested by the user is attempted immediately. If the attempt to reconfigure the system is successful, it is also stored in the cfgsh internal configuration “database.”

The user can ask to view cfgsh internal configuration database, which reflects (barring

[5]As can be guessed, the source for the suggested values for the timezone command are the list of files found in `/usr/share/zoneinfo/`. These are dynamically generated and are a good example of how cfgsh utilizes the GNU readline library to create a friendly user interface.

```
linbox>network
linbox (network)>help
interface   Enter interface configuration mode:
            interface [interface].
    route   Enter route configuration mode:
            route [priority].
  default   Display or set default gateway ad-
            dress: gateway [address].
 resolver   Enter domain name resolution con-
            figuration mode: resolver.
     exit   Return to root mode.
     quit   Logout.
     help   Display this text.
linbox (network)>
```

Figure 2: `cfgsh` network sub-menu

```
linbox>timezone
  timezone Display or set time zone: timezone [time zone].
Africa      Cuba        GMT+0       Kwajalein   Pacific     W-SU
America     EET         GMT-0       Libya       Poland      WET
Antarctica  EST         GMT0        MET         Portugal    Zulu
Arctic      EST5EDT     Greenwich   MST         ROC         iso3166.tab
Asia        Egypt       HST         MST7MDT     ROK         posix
Atlantic    Eire        Hongkong    Mexico      Singapore   posixrules
Australia   Etc         Iceland     Mideast     SystemV     right
Brazil      Europe      Indian      NZ          Turkey      zone.tab
CET         Factory     Iran        NZ-CHAT     UCT
CST6CDT     GB          Israel      Navajo      US
Canada      GB-Eire     Jamaica     PRC         UTC
Chile       GMT         Japan       PST8PDT     Universal
linbox>timezone Africa/Lu
  timezone Display or set time zone: timezone [time zone].
Luanda      Lubumbashi  Lusaka
linbox>timezone Africa/Lusaka
```

Figure 3: `cfgsh` timezone context sensitive help

bugs: see below on loosing sync with the system **??**) the system status using "show config" command. When used, the "show config" command will display the list of `cfgsh` commands that, once fed into `cfgsh`, will re-create the current configuration state.

Figure 4 shows an example of such a report.

In order to save the current system information for the next system boot, the user enters the command "save," which stores the configuration as a text file comprised of `cfgsh` commands. If issued, those commands will bring the system to the exact current state. This config text file looks exactly like the output of the "show config" commands (and is in fact generated from the same code).

```
linbox>show config
# Configuration Shell config file
hostname linbox
timezone Israel/IDT
network
    interface eth0
        dhcp off
        ip 192.168.1.1
        netmask 255.255.255.0
        broadcast 2192.168.1.255
        exit
    default none
    route 0
        set none
        exit
    route 1
        set none
        exit
    route 2
        set none
        exit
    resolver
        primary 194.90.1.5
        secondary 194.90.1.7
        search codefidence.com
        exit
    exit
role none
linbox>
```

Figure 4: cfgsh show config command output

Unless the user has issued the "save" command, all the changes to the system configuration are in effect only until the next system reboot, at which point the previous configuration will be used.

4.2 Setup mode

The purpose of setup mode is to allow `cfgsh` to set up the system as a replacement for system rc files. This mode is entered by running the program with the "setup" argument. Normally, this will be done once when the system boots, on every boot, by calling the program from the system `inittab(5)` file.

During setup mode, `cfgsh` reads the text config file saved using the "save" command in interactive mode and executes all of the command in the file in order to automatically set up the embedded system while also initializing the run time configuration data base in the shared memory segment for future instances of `cfgsh` running in interactive or silent mode.

After the file has been read and all the commands executed, `cfgsh` exists. When running in this mode, the normal prompt and some output is suppressed but normal messages are printed to stdout (e.g. "the system IP is now 192.168.1.1").

4.3 Silent mode

Silent mode is `cfgsh` way of supporting a simple programmable interface to manage the configuration to other programs, such as web management consoles and the like. This mode is entered by supplying the argument "silent" when running the program.

In this mode `cfgsh` runs exactly like in interactive mode, except that the prompt and some verbose output is suppressed. A program wishing to change the system configuration can simply run an instance of `cfgsh` running in silent mode and feed it via a Unix pipe `cfgsh` command for it to execute.

4.4 Internal configuration database

The internal configuration database is kept in a POSIX shared memory object obtained via `shm_open(3)` which is shared between all instances of `cfgsh`[6] and which stays resident even when no instance of `cfgsh` is running.

Thanks to this design decision, `cfgsh` does not need to re-read configuration files or query system interfaces when an instance of it is being run, since the information is available in the shared memory object.

[6]At the time of writing this paper, cfgsh still misses *correct* code that will prevent race conditions when accessing the shared memory area by multiple instances at the same time. This is however on the TODO list…

```
typedef struct {
  char ip[NUMIF][IPQUADSIZ];
  char nmask[NUMIF][IPQUADSIZ];
  char bcast[NUMIF][IPQUADSIZ];
  char gw[IPQUADSIZ];
  char ns_search[HOST_NAME_MAX];
  char ns1[IPQUADSIZ];
  char ns2[IPQUADSIZ];
  char role[PATH_MAX];
  char tz[PATH_MAX];
  char dhcp[NUMIF][DHCP_OPT];
  char dhcp_is_on[NUMIF];
  char hostname[HOST_NAME_MAX];
  char route[ROUTE_NUM][MAX_ROUTE_SIZE];
  char num_ifs;
} CONF;
```

Figure 5: Internal configuration database structure

This design also suffers from at least one downside: since most of the information in the configuration database is already present in the system in some form (the Linux kernel for IP addresses or /etc/resolv.conf for resolver address for example), there is always a risk of losing sync with the real state of the system. Despite this down side we believe that the central database which holds all the configuration information in a unified format is a design win (for embedded systems) despite the replication of information.

Figure 5 shows the structure of this internal database.

4.5 Command structure

`cfgsh` menus are comprised from arrays of commands. The program maintain a pointer to the current menu which is initialized in program start to the array of the main menu. Each choice of a sub-menu simply replaces the current menu pointer with the address of the ap-

```
typedef struct {
  char *name;
  rl_icpfunc_t *func;
  char *doc;
  complete_func_t *complete_func;
  char * complete_param;
} COMMAND;
```

Figure 6: `cfgsh` commands structure

propriate command array. It also updates the prompt.

Each command entry in the command array is a command structure which holds a pointer to the function to be called to perform the command, a description line to be shown as part of the help, a GNU readline library command competition function to perform context sensitive help for the command and a parameter to pass to the completer function to enable re-use of common functions (like directory competition).

Figure 6 shows the structure used to hold a single command entry.

4.6 Atomic configuration update

As have been described previously, `cfgsh` keeps the configuration database in memory and only commits it to disk (as a text file containing `cfgsh` commands) at the user requests via the "save" command. The same file is then used during the next boot to initialize booth the system and `cfgsh` own configuration database.

As can be understood, writing this configuration file correctly so that in to point on time we will not have a corrupt (or empty) configuration, is very important part of what `cfgsh` is meant to accomplish.

The method used is a very simple and well know one, which is based on the fact that the

```
int commit_file(char *tmp_file, char *file)
        int ret = 0;
        int fd = open(tmp_file, O_RDWR);
        if(fd == -1) return errno;
        if((ret = fsync(fd)) == -1) {
          close(fd);
          goto error;
        }
        if((ret = close(fd)) == -1) goto error;
        if ((ret = rename(tmp_file, file)) != 0)
                goto error;
        return 0;
error:
        unlink(tmp_file);
        return ret;
}
```

Figure 7: The commit_file() procedure

POSIX standard mandates that the if the second argument to the `rename(2)` system call already exists, the call will atomically replace the old file for the new file such that there is not point at which another process attempting to access the original name will find it missing.

To utilize this fact, we simply first created a full configuration file at a temporary location, sync its content to disk using `fsync(2)` and then `rename(2)` the new file over the old file.

Figure 7 shows the code of the `commit_file()` procedure that does the actual heavy lifting.

One thing which is perhaps missing from the procedure is a sync to the directory which holds the configuration file after the rename is over. Without this extra sync a sudden power failure after the rename may result in the directory entry never being written to permanent storage and the old configuration file used after reboot.

We believe that this is a valid scenario, as our purpose is to guarantee that the operation either fails as a whole or succeed as a whole but people who consider (quite rightfully) a system which boots with the previous IP address and network parameters after a configuration save a failure can simple add an fsync to the directory where the configuration file resides.

This brings up another issue to consider - the atomicity of this method is really only valid if and only if the underlying file system saves a directory entry as part of an atomic transaction. Since file systems that do exactly this are not rare (e.g. Ext3 [ext3]) this is considered a reasonable requirement by the authors, but it is worth noting by would be implementors.

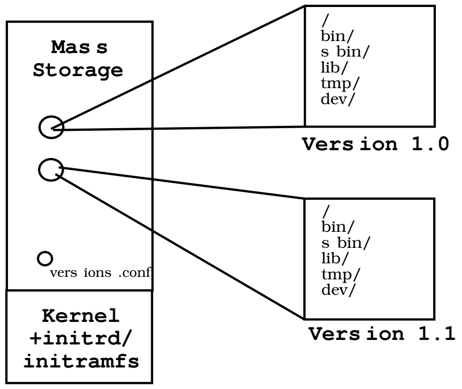
Mas s Storage
vers ions .conf
Kernel +initrd/ initramfs
/ bin/ s bin/ lib/ tmp/ dev/
Vers ion 1.0
/ bin/ s bin/ lib/ tmp/ dev/
Vers ion 1.1

Figure 8: File systems layout with sysup

5 `sysup` – embedded GNU/Linux system software update utility

`sysup`—an embedded GNU/Linux system software update utility—is a very simple tool that is meant to run at boot time from initrd/initramfs of a Linux-based embedded system in order to mount the root file system. Its objective is allow for an easily and atomically update-able embedded system software versions.

5.1 File system structure

To utilize sysup, the system file system layout must be done in a certain specific way, which is a little different from the normal layout used in embedded systems.

We shall define several different file system:

Main storage This is the file system on the main storage of the system—usually the flash disk. JFFS2 [JFFS2] or Ext3 appropriate file system types. This file system will contain configuration files and images of versions file system but not system software or libraries.

Version image This is a file system that contains the files for a specific version of the system software. It is meant to be used as the root file system of the system and contains all binaries, static configuration files, device files and software. Version images are (possibly compressed) loop back images and reside as files on the Main storage file system. Cramfs or Squashfs are the prime candidate as the type of these file system, although an Ext2 file system can be used as well if it is mount read-only.

initrd/initramfs This is a file system image or cpio archive which are used to host the files of the sysup mechanism. These file system are mounted during boot and discarded once the boot sequence is complete. Again, Cramfs, Squashfs, or Ext2 are good choices for the kind of this file system.

Figure 8 shows a schematic of the various file systems layout in relation to each other.

5.2 The boot process with sysup

What sysup does can be best explained by describing the basic boot process on a sysup enabled system:

1. System boots.
2. Boot loader loads kernel and initrd/initramfs images into memory.
3. Kernel runs and mounts initrd or initramfs content
4. sysup is run.
5. sysup mounts the main storage.
6. sysup locates on the main storage the versions.conf file.
7. sysup locates on the main storage a version image.
8. sysup takes an MD5 signature of the version image and compares it to the one stored in the versions.conf file.
9. If the signatures do not match or in response to any other failure, sysup rolls back to the previous version by moving on to the next entry in the versions.conf file and branching back to stage 7[7].
10. If the signatures match, sysup will loop back mount the version image in a temporary mount point.
11. sysup will move the mount point of the main storage device into a mount point in temporary mount point of the version image. This is done using the "new" (2.5.1, but back ported to 2.4) MS_MOVE mount flag to `mount(2)`[8].
12. sysup will then `pivot_root(2)` into the temporary mount point of the mounted version image, thus turning it to the new root file system.

[7]At the time of the writing of this paper only 2 versions.conf entries are supported, but changing this is very easy should the need ever arise.

[8]Used by the `-move` options to `mount(8)`.

```
7e90f657aaa0f4256923b43e900d2351 \
/boot/version-1.5.img
2c9d55454605787d5eff486b99055dba \
/boot/versions-1.6.img
```

Figure 9: versions.conf

13. The boot is completed by un-mounting the initrd or initramfs file systems and execing into `/sbin/init`.

An example version.conf file is shown in figure 9. A very simple implementation of sysup as a shell script is in figure 10.

5.3 System software update

The above description of a sysup boot sequence sounds more complicated then usual. On the other hand, the system software upgrade procedure is quite simple:

1. Download a new version image to the main storage storage.
2. Calculate its MD5sum and do any other sanity checks on the image.
3. Create a new versions.conf file under a temporary name, with the MD5 and path of the new version image as the first entry and the old version image and its MD5 sum (taken from the current version.conf file) as the second entry.
4. fsync the new versions.conf under its temporary name.
5. rename(2) the new version.conf file over the old one.
6. Reboot.

```
#!/bin/sh
# Name and path to file with filename and MD5s
VERSIONS=versions
# How to get the first line of the file
LINE=‘tail -n 1 versions‘
# File name if MD5 is correct, empty otherwise
FILE=‘./md5check $LINE‘
# How to get the second line of the file
ALTLINE=‘head -n 1 versions‘
# File name if MD5 is correct, empty otherwise
ALTFILE=‘./md5check $LINE‘
# File system type of images
FSTYPE=ext2
# Mount point for new root
MNT=/mnt
# File system type for data parition
# (which holds the image files)
DATAFSTYPE=ext3
# Mount point of data partition
DATA=/data
# Name of directory inside the images
# where the original root mount point
# will be moved to
OLDROOT=initrd
# device of data parition
DATADEV=/dev/hda1
# Emergency shell
EMR_SHELL=/bin/sh
boot() {
        mount -t $FSTYPE $FILE $MNT && \
        cd $MNT && \
        pivot_root . $OLDROOT &&
        mount $OLDROOT/$DATA $DATA -o move && \
        umount $OLDROOT && \
        exec /sbin/init
}
mount -t proc /proc && \
mount -t $DATAFSTYPE $DATADEV && \
if test -z "$FILE"; then \
   echo "Attempting to boot 1st choice" && boot(); \
fi && \
if test -z "$ALTFILE"; then \
   echo "Attempting to boot 2nd choice" && boot(); \
fi
echo "Boot failure." && exec $EMR_SHELL
```

Figure 10: sysup shell script

Once again, just like with `cfgsh` configuration file, the POSIX assured atomicity of the rename(2) system call, guarantees that at no point in time will a loss of power lead to a system that cannot boot.

5.4 Variations on a theme

To this basic scheme we can add some more advanced features as described in this section. None of these implemented in the current version of sysup, but they are on our TODO list for future versions.

5.4.1 Upgrade watchdog

A version image might have good checksum and mounted correctly, but the software in it might be broken in such a way as to get the machine stuck or reboot before allowing the user to reach a stage that he or she can roll back to the previous version.

To resolve this issue, or at least to mitigate its effect to some extent, the following addition can be made to the sysup boot process:

- During boot, before mounting a version image file, sysup should look on the main storage file system for a “boot in progress” indicator file. If the file is there, it should roll back and boot the next entry of versions.conf file.
- If the file is not there and before sysup mounts the new version image file, it will create a “boot in progress” indicator file on the main storage file system.
- After a version image finished its boot successfully to such a degree that the user can request a software version upgrade or

downgrade, the software on the version image will delete this "boot in progress" indicator from the main storage file system.

This addition to the boot process allows detect errors that would otherwise lead to a system that reboots into a broken version in an infinite loop.

5.4.2 Network software updates

Another possible extension to the sysup boot model is to extend sysup to look for newer version to boot in a network directory of sorts, in addition to the `versions.conf` file.

If a newer version is found, it is downloaded and Incorporated into the regular version repository on the main storage (perhaps deleting an older version to accommodate).

If the newest version on the network directory is the same as the version stored on the mass storage, boot continues as before.

5.4.3 Safe mode

Sometime, despite our best efforts, the version images on the flash can become corrupted. In such an event, it can be very useful to allow the sysup code in the initrd/initramfs image, when it detects such an occurrence, to enter a "safe mode" which will allow the minimal configuration of the system (e.g. network settings) and download of a fresh version image to flash.

5.5 The Achilles heel of sysup: kernel upgrades

The reason sysup is able to supply atomic upgrade of software version is exactly because, thank to the ability of the Linux kernel to loop back file system images, all the system software can be squeezed into a single file. Unfortunately, the kernel image itself cannot be included in this image for obvious reasons, which leads to a multitude of problems

As long as we are willing to treat the kernel and the initrd/initramfs images with it, as a sort of a boot ROM, only update-able in special circumstances by re-writing in a non atomic fashion the appropriate flash partition, we have no problem.

Unfortunately, this is not always enough. Bugs in the kernel, support for new features and the need of kernel modules to be loaded into the same kernel version for which they were compiled may require kernel upgrades, not to mentions bugs in sysup code itself. . .

There are two ways to overcome this limitation, each with its own set of problems:

5.5.1 Two kernel Monte

Under this scheme, we add another field to the `versions.conf` file—the version of the kernel required by that version image. sysup then needs to check whether the currently running kernel is the right one. If it is, we proceed as usual. If it is not we utilize a Linux based Linux loader such as kexec or similar in-kernel loaders [kexec] [9] and boot into the correct kernel. This time we will be in the right kernel version and boot will continue as normal.

This method works quite well, however it has two major drawbacks:

- At the time of the writing of this paper, neither kexec, two kernel Monte or lobos

[9] Our tests with this mode of operation were done with the Two kernel Monte module from Scyld Computing.

are integrated into the vanilla kernel, requiring a patch.

- Those solutions that do exist seems to cover x86 and to some degree ppc32 architecture only.
- Using this approach lengthens boot time.

5.5.2 Cooperating with the boot loader

As an alternative to booting a Linux kernel from Linux, we can use the very same mechanism discussed before of marking each version with the required kernel version to run it and simply change the boot loader configuration to boot the right kernel version next time and then reboot the machine. If all goes well, when we next run, we will already be running under the right kernel.

The drawback of this method is of course that we are now tied to the specific feature set of a specific boot loader and not all boot loader lend themselves easily to this sort of cooperation with regard to choosing the Linux kernel image to boot.

6 Read, Copy, Update

One of the readers of the early draft of this paper remarked how much our approach to create atomic update of complex data by creating a copy and then switching pointers to this data is similar to the well known IBM patented RCU method utilized in the latest Linux kernel versions.

While we happily agree that the mechanism is the basically the same, we would like to point out that the purpose of applying the technique (which we of course do not claim to have invented) is different: the RCU implementation in the Linux kernel is done to avoid locking when accessing data structure as an way to speed up access to these data structures, where as our use of the technique is done because it is *impossible* to lock the data structure we want to access, barring the use of a battery attached to each embedded device.

It is interesting though, to note the usefulness of the same technique to solve different, but related problems.

7 Future directions

Apart from implementing our lengthy TODO list, some of which has been discussed in this paper, there are some “blue skies” areas of interest for additional research with `cfgsh` and sysup.

The most interesting one, in the humble opinion of the authors, is the possibility that the techniques outlined here and implemented in the two projects can be useful outside the scope of embedded systems design, especially with regard to “stateless Linux,” virtual machine settings and GNU/Linux-based clusters.

Because the approach presented here essentially locks all the information about software versions and configuration in a couple of easily controlled files, and supports transactional management of these resources it is hoped that developers and researches working in those fields would be able to utilize the easy ability to change and save the state of a machine with regard to software version and configuration to create mechanism to automatically and safely control their systems, virtual instances or cluster node in the same way that we demonstrated can be done with embedded systems.

8 Thanks

The authors wish to thank the following people:

To Orna Agmon, Oleg Goldshmidt and Muli Ben-Yehuda for their support and advice during the writing of this paper.

To Aviram Jenik and Noam Rathaus from BeyondSecurity Ltd. for sponsoring the original creation of `cfgsh` as Free Software licensed under the GPL (which is why I forgive them for forgetting to send me their patches to the program.)

Last but not least, to my wife Limor, just for being herself.

References

[Squashfs] Artemiy I. Pavlov, *SquashFS HOWTO*, The Linux Documentation Project, `http://www.artemio.net/projects/linuxdoc/squashfs/`

[ext2] Remy Card, Theodore Ts'o, Stephen Tweedie *Design and Implementation of the Second Extended Filesystem*, The Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9, `http://www.artemio.net/projects/linuxdoc/squashfs/`

[Murphy] `http://dmawww.epfl.ch/roso.mosaic/dm/murphy.html#technology`

[GNU Readline] Chet Ramey and others, `http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html`

[cramfs] Linus Torvalds and others,

[ext3] Stephen Tweedie, *EXT3, Journaling Filesystem*, The Proceedings of Ottawa Linux Symposium 2000, `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`

[JFFS2] David Woodhouse, *JFFS: The Journalling Flash File System*, `http://sources.redhat.com/jffs2/jffs2.pdf`

[Embedded Linux Systems] Karim Yaghmour, *Building Embedded Linux Systems*, O'Reilly Press, ISBN: 0-596-00222-X

[kexec] Andy Pfiffer, *Reducing System Reboot Time With kexec*, `http://developer.osdl.org/rddunlap/kexec/whitepaper/kexec.html`