Administration and Deployment of Wireless Mesh Networks

Masterarbeit
der Philosophisch-naturwissenschaftlichen
Fakultät der Universität Bern

vorgelegt von

Daniel Balsiger
April 2009

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The necessity for wireless communication is gaining in importance more and more these days. Wireless Mesh Networks (WMNs) are a promising technology in the domain of radio communication.

The Administration and Deployment of Adhoc Mesh (ADAM) framework provides management and deployment support for WMNs. It consists of three main parts. First, the ADAM build system simplifies the creation of appropriate software images for wireless mesh nodes. Second, the ADAM configuration framework provides robust and safe configuration and software updates during the whole life cycle of a Wireless Mesh Network (WMN). The third optional component is the ADAM Graphical User Interface (GUI) which simplifies monitoring and management tasks for a WMN by providing assistance with a graphical web interface. The ADAM GUI is developed by Simon Morgenthaler during his Bachelor thesis and therefore not part of this Master thesis.

This thesis is divided in eight chapters. This chapter introduces WMNs and the supported hardware devices. In Chapter 2 general requirements for the software for embedded devices are shown. Chapter 3 focuses on related work and analyses existing solutions for software compilation and management of WMNs. The implementation of the ADAM build system is described in detail in Chapter 4. The different image types created by the ADAM build system are explained in Chapter 5. The implementation of the ADAM configuration framework is described in detail in Chapter 6. Chapter 7 focuses on the evaluation of the ADAM framework. The conclusion of this thesis is shown in Chapter 8.

1.1 Wireless Mesh Networks

A WMN is a network consisting of multiple nodes, which are capable of communicating through radio devices. An example of a WMN is shown in Figure 1.1.

WMNs are a possible solution for communication in regions, like deserts or mountains, where development of a wire based communication is too expensive or just not possible. WMNs can also be used for communication in well developed

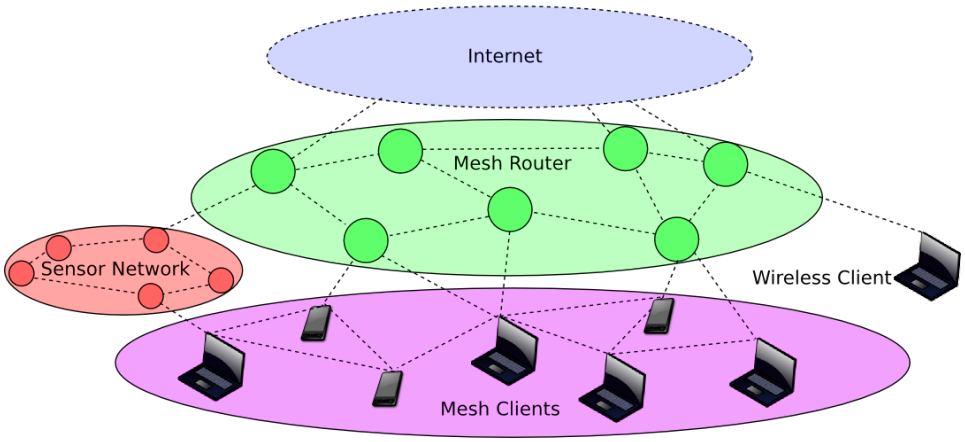


Figure 1.1: Example of a hybrid Wireless Mesh Network

areas since they can provide a cost efficient variant of accessing other networks. For example a WMN could be used as an interconnection network to various other networks like the Internet or local sensor networks.

In comparison to centralised wireless communication forms, like for example Global System for Mobile communications (GSM), the nodes of a WMN are fully self-contained and need no base station or access point. WMNs can use special mesh routing mechanisms and protocols which can enhance the stability and fault tolerance of the network by providing intelligent multipath routing algorithms. A failure or temporary loss of connectivity of some nodes should not cause a failure of the network, other reachable nodes should take over the work seamlessly.

The wireless communication protocol according the Institute of Electrical and Electronics Engineers (IEEE) 802.11 can be used for implementing a WMN. WMNs generally use the 802.11 adhoc mode for the interconnection of the individual nodes. The 802.11 adhoc mode allows communication between multiple nodes without the need for an access point. In order to support normal clients which are not adhoc capable, a mesh node may further provide a standard infrastructure network like an ordinary wireless access point.

According to [1], WMNs get classified into client, infrastructure and hybrid WMNs. Infrastructure WMNs provide a wireless mesh backbone network for conventional clients, which are not capable of mesh routing themselves. Client WMNs on the other hand are classical peer-to-peer networks among multiple clients without the need for an access point. Hybrid WMNs are a combination of both client and infrastructure WMNs and therefore the most general type of a WMN. Clients are either connected by the conventional infrastructure mechanisms or by additional mesh mechanisms on the clients. The implemented framework focuses on such hybrid WMNs. Other papers, which provide an overview on the topic of WMNs are [2], [3] and [4].

Radio communication is influenced by weather conditions, static or dynamic obstacles and interferences with other local radio signals in the environment. Nodes may therefore be temporary unreachable and sometimes additional nodes have to be added to the network to enhance throughput or connectivity.

1.2 Hardware

The hardware for setting up a WMN depends on the environment where the deployment takes place. Nodes which are deployed at hardly accessible, isolated places, like trees or rooftops, have to be protected against weather influences and theft, while nodes placed in buildings do not need such protections. In many scenarios the power consumption of the device is also crucial as even fully solar powered nodes are imaginable. Another important factor of WMN hardware is the price per node. For all these reasons hardware for wireless mesh nodes is generally realised in an embedded form factor. Embedded systems are computer devices designed to perform one or a few dedicated functions. Normally Central Processing Unit (CPU) power and the amount of Random-Access Memory (RAM) and secondary storage is limited in comparison to contemporary workstations. Some embedded devices are realised in a System on a Chip (SoC) and contain a CPU, RAM, secondary storage and even communication controllers in only one single integrated circuit. A WMN may also be set up of different hardware devices, designed for different locations, transmissions power requirements, number of network interfaces or other attributes according the function of the involved nodes. Such a network is usually called a heterogeneous WMN.

Being hardware independent, the ADAM build system supports currently several platforms, with different processors, wireless chipsets and secondary storage. In the following a list of hardware devices is shown, for which a working software image has been built with ADAM. Some of these devices require special kernel sources, others only use a different wireless driver. The implemented ADAM configuration framework supports heterogeneous WMNs containing the hardware platforms, whose hardware attributes and features are shown in Table 1.1.

- The Meraki Mini [5], shown in Figure 1.2

  The important hardware attributes of the Meraki Mini board are shown in Table 1.1. This platform is used for the ADAM evaluation tests described in Chapter 7.

- The PC Engines Alix [6], shown in Figure 1.3

  The hardware attributes of the Alix board are shown in Table 1.1. Like the Meraki, the Alix board is used for the ADAM evaluation tests described in Chapter 7.

Platform	Meraki Mini	Alix	WRAP
Vendor	Meraki Inc.	PC Engines GmbH	PC Engines GmbH
CPU	MIPS 4KEc	AMD Geode	NSC Geode
MHz	180	500	233
Architecture	MIPS	i386	i386
RAM	32 MByte	256 MByte	128 MByte
Ethernet Port	1	1	1
Wireless	Atheros 2315	2x Mini PCI (Atheros)	2x Mini PCI (Atheros)
Storage	8 MByte NAND Flash	Compact Flash	Compact Flash
Expansion	None	USB 2.0	USB 1.1

Table 1.1: Supported hardware platforms

- The PC Engines Wireless Router Application Platform (WRAP) [7], shown in Figure 1.4

  The WRAP board is the predecessor of the Alix board. WRAP and Alix boards are very similar in their hardware design. The important hardware attributes of the WRAP board are shown in Table 1.1.

- The Neo Freerunner mobile phone (GTA02) by Openmoko [8]

  This device uses an Advanced RISC Machine (ARM) processor. The Neo Freerunner phone is used for testing the support for ARM processors in ADAM. Special kernel sources published by the vendor under an open source license are required to build a working software image.

- The Fusiv VX180_IFE6 VDSL evaluation board by Ikanos [9]

  This device uses a Microprocessor without Interlocked Pipeline Stages (MIPS) compatible processor. A partially closed source kernel provided by the manufacturer is required to build a working software image. The Fusiv VX180 is used for testing the support for external kernel sources provided by device manufacturers.

- Several i386 compatible Thinkpad notebooks by IBM/Lenovo

  These devices use an i386 compatible processor and are used to test the support for additional wireless chipsets (e.g. Cisco Aironet 340, Intel WiFi Link 5300AGN) in ADAM.

- Wireless routers (63XX chipset based) by Broadcom

  These devices use a MIPS compatible processor and a Broadcom wireless chipset. Therefore the routers by Broadcom are used for testing the support for 63XX based wireless chipsets in ADAM.

Figure 1.2: Meraki Mini node hardware with indoor case

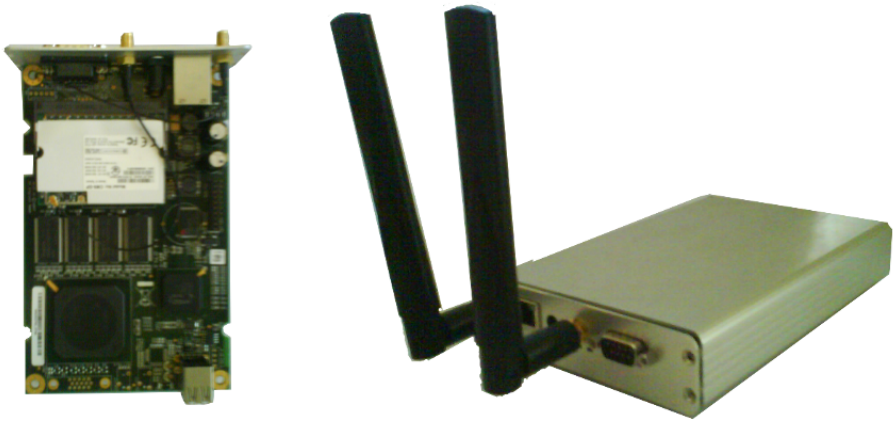


Figure 1.3: PC Engines Alix node hardware with indoor case

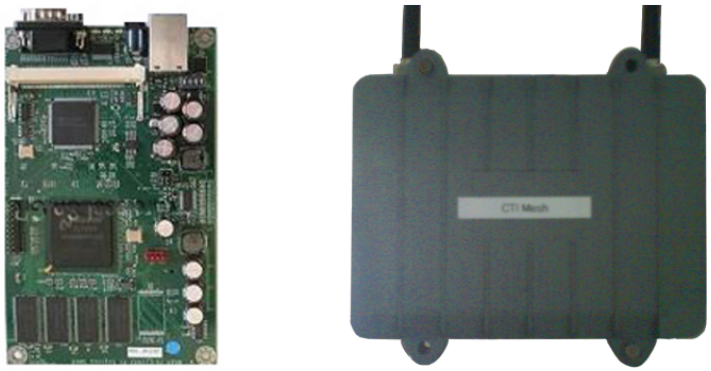


Figure 1.4: PC Engines WRAP node hardware with outdoor case

1.3 Software

Since mostly embedded systems are used as hardware platforms for wireless mesh nodes, software for these nodes has to be tailored for the use on such devices. Main limitations for the software are actually the amount of RAM and secondary storage of the hardware devices involved. If a general purpose software framework for a WMN is envisioned, a portable and architecture independent framework is required, which allows supporting a wide variety of available hardware devices. Due to the lack of hardware requirements as enough RAM, secondary storage or computing power, the software for many of the involved embedded devices has to be compiled on other systems which fulfil these requirements. Another difficult task is the debugging of software for embedded systems due to space limitations, for example if debugging symbols were stripped out of the binaries or no debugger can be installed on the node platform.

1.4 Configuration

WMNs can be dynamic, nodes can temporary appear and disappear and routes from a given source to a desired destination in a WMN can change over time. Therefore a robust configuration mechanism which handles dynamic changes as well as temporary outages is required. It should provide functions for monitoring and configuration of the different parameters of the network, like the reachable nodes, the software installed and network addresses and routing mechanisms used on the nodes. A configuration framework used for WMNs should provide functions for monitoring and configuration of the different parameters of the network, like the reachable nodes, the software installed and network addresses and routing mech-

anisms used on the nodes. Moreover features like software upgrades, configuration changes and the possibility to define new configuration parameters should be supported too. In addition to normal mesh nodes the provided ADAM framework supports some distinguished nodes which have some additional management capabilities used for configuration and monitoring purposes. These node are called management nodes throughout this document.

Chapter 2

Software for Embedded Systems

2.1 Linux on Embedded Systems

Many embedded systems today exist today, e.g. mobile phones, car navigation devices and wireless nodes. Many of them use a proprietary operating system, often designed for a specific device of a specific company. Such operating systems are mostly in binary formats and seldom customisable.

The fact, that Linux [10] supports many hardware architectures and its open source license makes it a very good candidate for a general purpose operating system. Additionally developing software for Linux and its related GNU is Not Unix (GNU) tools and libraries is a standardised process and very well documented. Most of the software used is written with a high degree of portability in mind. Therefore, Linux is a very good choice of an operating system for wireless mesh nodes, if the target platform has a minimal amount of RAM and secondary storage. Most node devices used in a WMN meet these conditions. Nowadays some manufacturers ship their embedded devices already with a Linux firmware and profit therefore from the mentioned facts. Their number is continuously growing, Linux is already running on several mobile phones, handhelds and many other wireless devices available today.

2.2 Cross Compiling Software

Generally it is a complex task to compile and link binaries on a Linux platform, even for the machine the compiler runs on. Such compilers are called native compilers. For their proper functionality many preconditions must be met and many tools with appropriate versions have to be installed (shell, assembler, linker and various libraries). In addition, some software packages often require other software packages in a predefined version, which even enhances the complexity.

Since software for WMNs mostly runs on embedded devices it is often not a good choice to use a native compiler on these devices due to space and performance limitations. On some small devices it is often not even possible to install a native

compiler and the corresponding tools. To compile software for such devices cross compilers are used.

A cross compiler is a compiler which runs on machine A (called host) and generates code for another machine B (called target). This is exactly what is needed, when it comes to compile software for embedded devices. There even exist cross compilers which are built on machine A (called build), run on machine B (called host) and produce code for machine C (called target). Such compilers are called canadian cross compilers and are mentioned only for completeness. In the normal cross compilation procedure used by the ADAM build system, the host and build platform are the same and called host throughout this document.

2.3 Requirements for a Build System

A build system for embedded wireless mesh nodes should at least fulfil the following requirements:

- WMNs are often heterogeneous and therefore consist of more than one node hardware platform. The build system should be able to support as many target platforms as possible.
- The developer wants to choose the host platform used. Therefore the build system should be able to support as many host platforms as possible.
- The effort of the developer to use the build system should be as small as possible, so it has to be easy to understand and well documented.
- A network evolves and maybe new node hardware is added in the future. The build system should have capabilities to build the already deployed software for the new hardware, and the effort for this adaptions should be as minimal as possible.
- The selection of the software which is built and installed should be left to the developer. The build system should use as few mandatory software as possible.
- The compiled software as well as the system configuration should be in a format which is easy to deploy to a wireless network consisting of multiple nodes with different target architectures. This leads to some sort of software images for the wireless nodes, unless a package manager is used.
- The configuration of wireless mesh nodes is dynamic. Therefore the build system has to respect the configuration mechanism of the nodes during the installation process.
- All nodes in a network should provide the same minimal functionality. Therefore the software running on the target nodes should not be dependent

on the hardware platform. To guarantee a maximum degree of compatibility between all nodes, they all should use the same software in the same version.

2.3.1 Cross Toolchain

As seen before, a cross compiler has to be used for compiling source code for the target platform. The collection of the cross compiler, its related binary tools and the C library as well as the system headers for the target platform is called a cross toolchain.

Since multiple target and host platforms have to be supported, it is definitely not a good choice to provide cross toolchains in binary format for all possible combinations of target and host platforms. This leads to create the cross toolchain for the actually desired combination of host and target platform from source. If the final software used on the nodes has to be dynamically linked against shared libraries, which is generally the case on Linux platforms, the creation of the cross toolchain has to accommodate this. The following steps outline the normal way to create a cross toolchain.

1. Installation of operating system headers

   Operating system headers define data structures (e.g. machine endianness) and system procedures (e.g. system calls). For cross compilation they have to be copied to the right place, which is no problem in general.

2. Installation of machine-specific Executable and Linkable Format (ELF) binary tools

   The GNU binutils package provides utilities which are used for creating, inspecting, manipulating and linking ELF binaries. ELF binaries are the default binary type used on Linux platforms and are dependent of the target platform used. The normal compiler installed on the host platform is used to compile these tools for the target platform. After compilation the ELF binary utilities have to be installed to the right place.

3. Installation of an intermediate cross compiler

   The final goal is to create a cross compiler which is able to produce code, that is dynamically linked to the C library of the target platform. Since no C library for the target architecture has been installed yet, the C library must be cross compiled first. Therefore the intermediate cross compiler is used. For its compilation again the normal compiler installed on the host platform and the previously installed machine-specific ELF binary tools are used.

4. Installation of the target C library

   With the previously installed intermediate cross compiler the C library for the target platform is compiled and installed.

5. Installation of the final cross compiler

   Since a C library for the target platform is available now, the final cross compiler can be built. The final cross compiler is compiled by the intermediate cross compiler and replaces the intermediate cross compiler after its installation. The final cross compiler is able to produces code that is dynamically linked to the previously installed C library and is used to compile all further dynamically linked software for the target platform.

2.3.2 Compiler

The GNU Compiler Collection (GCC) [11] is a very widely used compiler suite and is the standard on Linux platforms. The build system uses the C compiler and optionally the C++ compiler of GCC. Unlike other cross compiled software, the cross compiler itself is never installed to a target device, therefore the size of the compiler binaries is not relevant for the storage footprint of the target device.

2.3.3 C Library

Another important question is which C library has to be used on the target platform. There are many implementations of C libraries nowadays. The common choice for Linux systems is the GNU C library (Glibc) [12]. However, for embedded systems with a small amount of secondary storage or RAM a smaller implementation of a C library is often more useful or even required.

The ADAM build system uses therefore the uClibc [13] C library which is a very small C library implementation (about 400 KByte on a i386 system), designed specially for embedded systems. The goal of uClibc is to be as small as possible, while staying compatible with Glibc in the most cases. Source code adjustments for using uClibc as replacement for Glibc are not needed in general or are relatively easy to achieve.

2.3.4 C++ Support

If the target board has to support software written in C++, generating a final C++ cross compiler and installing a standard C++ library is required during the installation of the cross toolchain. GCC already supports these tools and libraries and the required installation procedure is well documented. But C++ support involves more dependencies on the host platform than supporting only the C programming language in the cross toolchain. Therefore the C++ support is disabled in the default configuration of the build system, but it can be enabled easily by editing the default package selection for a specific target board.

2.4 Software Packages for the Target Platform

As seen before, it is useful to use the same software versions on all target platforms to ensure a high degree of compatibility between the involved target platforms. In addition, it is advisable to use as much of the normal software packages found on a non-embedded Linux platforms as possible, to reach even a higher degree of compatibility. Problems with this strategy are possible hardware limitations (mostly in secondary storage and RAM), which do not allow using the standard tools. Since all platforms should run the same minimal software, the selected software package has to be adjusted to the minimum amount of RAM, secondary storage and CPU power of all occurring supported target platforms.

Normally the software installed on Linux systems is grouped into so called packages. A package is a collection of software for a dedicated purpose, it can be a program, a library or only documentation. Modern Linux distributions generally provide software packages in binary form, which means that the compiled programs and libraries are collected in different packages which can be installed to or removed from the system at run time.

As far as open source software is concerned, these packages are available in source code format. The source code of a software package is generally available in compressed tar archive format downloadable from the website of the developer. The build system uses only these source code software packages, as the software has to be compiled with the corresponding cross compiler and linked against the special C library for the target platform.

2.4.1 Storage Limitations

Due to limitations in secondary storage or RAM, the software packages used on the nodes have to be small enough even for fitting on the node platform with the smallest amount of secondary storage and RAM. Nevertheless they should stay as compatible to the standard packages used on non-embedded Linux systems as possible. To achieve both of these contradictory requirements, some trade-offs concerning the selection of software packages have to be made. All the software components have to be carefully selected and alternatives to standard packages with a similar functionality but a smaller memory footprint are preferred. For example the C library used by the ADAM build system, uClibc described in Section 2.3.3, seems to be a good compromise between size, functionality and compatibility.

Another important package to mention in this context is Busybox [14], which provides a single, small multifunctional binary as replacement or the most standard UNIX tools, like e.g. sh, cp, mv, grep, sed, and awk. The resulting Busybox binary is very small (about 800 KByte on i386) compared to the collection of the replaced standard UNIX tools (about 5 MByte on i386). Newer versions of Busybox can even provide much more functionality than just these common tools. ADAM uses many of these additional features of Busybox. A complete list of the functions provided by the current Busybox binary used on the nodes is shown in Listing 2.1.

Another space relevant piece of software are the kernel modules. Normally, kernel modules are stored in binary form, but the use of the module-init-tools package makes it possible to store them in gzip compressed format, which reduces their storage footprint on the target system significantly.

```
root@meraki0:~ # busybox --help
BusyBox v1.11.2 (2009-02-03 20:48:52 CET) multi-call binary
Copyright (C) 1998-2008 Erik Andersen, Rob Landley, Denys Vlasenko
and others. Licensed under GPLv2. See source distribution for full notice.

Usage: busybox [function] [arguments]...
   or: function [arguments]...

     BusyBox is a multi-call binary that combines many common Unix
     utilities into a single executable.  Most people will create a
     link to busybox for each function they wish to use and BusyBox
     will act like whatever it was invoked as!

Currently defined functions:
     [, [[, adjtimex, ar, arp, ash, awk, basename, brctl, bunzip2, bzcat,
     bzip2, cal, cat, chgrp, chmod, chown, chpasswd, chroot, cksum, clear,
     cmp, comm, cp, cpio, crond, crontab, cryptpw, cut, date, dc, dd, df,
     dhcprelay, diff, dirname, dmesg, dnsd, du, dumpleases, echo, egrep,
     env, ether-wake, expand, expr, fakeidentd, false, fgrep, find, fold,
     free, fuser, getty, grep, gunzip, gzip, halt, head, hexdump, hostid,
     hostname, hwclock, id, ifenslave, init, install, ipcalc, kill,
     killall, killall5, klogd, last, length, less, ln, logger, login,
     logname, logread, losetup, ls, lzmacat, md5sum, mdev, microcom,
     mkdir, mkfifo, mknod, mktemp, mount, mountpoint, mv, nameif, nc,
     netstat, nice, nmeter, nohup, nslookup, od, passwd, patch, pgrep,
     pidof, pkill, poweroff, printenv, printf, ps, pscan, pwd, readlink,
     realpath, reboot, renice, rm, rmdir, sed, seq, sh, sha1sum, sleep,
     sort, split, start-stop-daemon, stat, strings, stty, su, sum,
     switch_root, sync, sysctl, syslogd, tac, tail, tar, tcpsvd, tee,
     telnet, test, tftp, tftpd, time, top, touch, tr, traceroute, true,
     tty, udhcpc, udhcpd, udpsvd, umount, uname, unexpand, uniq, unlzma,
     unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, watch,
     watchdog, wc, which, who, whoami, xargs, yes, zcat

root@meraki0:~ #
```

Listing 2.1: Supported functions of Busybox

2.4.2 Used Software Packages

Besides uClibc, Busybox and the Linux kernel, the build system cross compiles additional software packages for the nodes. Some of these packages are tailored specially for embedded devices, others are standard packages, which can be found on a normal, non embedded Linux system. As mentioned before, the question which package to choose is mostly a trade-off between space and functionality.

Sometimes it is even desirable to have the choice between multiple packages for the same purpose. A good example for such a package of choice is the web server used on normal nodes and the one used on management nodes. The one on management nodes has to support external scripting languages as PHP Hypertext

Package	Version	Intended purpose
Binutils	2.18	ELF binary tools
GCC	4.2.4	C and C++ cross compilers
uClibc	20080913	C library for embedded devices
Linux	2.6.26.8	Linux kernel
Madwifi	svn3380	Linux drivers for Atheros wireless chipsets
Zlib	1.2.3	General purpose compression library
Openssl	0.9.8j	SSL and cryptographic library
Curl	7.18.2	Networking library and URL tools
Busybox	1.11.2	Small replacement for common UNIX tools
Iproute2	2.6.26	Linux networking and traffic control tools
Wireless-tools	29	Linux wireless tools
Module-init-tools	3.4	Linux module utilities
Iputils	s20071127	Additional IPv4 networking tools
Ipv6calc	0.71.0	IPv6 address calculator
Radvd	1.2	IPv6 routing advertisement daemon
Iptables	1.4.0	Linux IPv4 and IPv6 netfilter tools
Dropbear	0.51	SSH client/server for embedded devices
Openntpd	3.9p1	NTP client/server
Nostromo	1.9	SSL and IPv6 single process web server
Sudo	1.6.9p17	Superuser rights manager
Db	4.4.20	Berkeley database needed for Cfengine
Cfengine	2.2.9	Distributed systems configuration engine
Olsrd	0.5.6	Mesh network routing daemon

Table 2.1: Default selection of software packages

Parser (PHP) or Python, while the one running on normal nodes, should be as small as possible and requires only a subset of the functionality. The default selection of software packages installed on a normal node, the versions used and the intended purpose of the packages is shown in Table 2.1.

Chapter 3

Related Work

In Chapter 2 the requirements of a build system for embedded devices and the need for a special cross compilation toolchain were outlined. The ADAM software framework tries to cover more than just being a build system. The integration of the image creation process as well as the configuration framework are further important aspects. The three main functions of ADAM can be described as follows.

- Customisable cross compilation build system.
- Creation of adequate software and configuration images and their deployment to multiple node platforms.
- An enhanced version of the management framework based on Cfengine [15] and its seamless integration in the build and deployment process.

This chapter focuses first on available cross compilation build systems for embedded devices. Second, existing management solutions for WMNs are analysed. In a third part the key features of the existing build systems and the management frameworks are discussed. The last section describes assets and drawbacks of the Secure Remote Management and Software Distribution for Wireless Mesh Networks (SRM) [16] framework, which can be regarded as predecessor of ADAM.

3.1 Existing Build Systems

In this section three different existing cross compilation build systems are analysed in terms of complexity, available documentation and compatibility with the above requirements. Furthermore, the limitations of the existing solutions are shown.

3.1.1 OpenWrt

OpenWrt [17] is a Linux distribution tailored for embedded devices. The OpenWrt distribution provides a package manager based approach for installing software on embedded devices. Creating a static single firmware is not intended by this

distribution, instead a root file system on secondary storage is used. OpenWrt is normally provided in binary format, but can be built also fully from the available source due to its open source license. For this reason, a cross compile environment is provided, which can build and compile the OpenWrt distribution for a specific device.

OpenWrt is able to cross compile many software packages for a wide variety of embedded wireless devices. OpenWrt uses the small C library uClibc [13] which is also desirable for ADAM. The OpenWrt team provides required adjustments to source code in form of patch files. They contain mostly hardware platform dependent changes for the Linux kernel, which allows using Linux on many devices. Other patches contain adjustments and bug fixes for the cross toolchain installation, mostly related to uClibc. Some of these patches are also needed by ADAM (e.g. Linux patches for the Meraki node). The build system of OpenWrt is controlled by various Makefiles and the documentation for the most features of the build process is available.

However, OpenWrt does not match the criteria of being compatible with the ADAM image creation and management framework. OpenWrt relies on its package management system when creating firmware for particular devices. The package manager based approach installs packages to a read and writable file system on the device's secondary storage. Therefore less software can be installed on small devices, whose amount of RAM is bigger than the amount of secondary storage compared to the compressed read-only all-in-one software image used by ADAM. The separation of cross compiled software and configuration data is another needed feature of ADAM. Package manager based solutions rather do the opposite, they combine the software and its configuration in one package. The effort of adjusting this complex package manager based approach for the needs of the image creation process (e.g. divided software and configuration) would have been much more time consuming, then developing a new build system. However, many ideas and the already mentioned patches provided by the OpenWrt team were used during the development of the ADAM build system.

3.1.2 Openembedded

Openembedded [18] is an open source cross compile environment. It is designed to create a complete Linux distribution for embedded systems. Openembedded offers the possibility to cross compile a huge amount of software packages. Mostly every software package can be cross compiled with the help of Openembedded.

Moreover, every aspect in Openembedded can be configured, the compilation procedure as well as the selection of the packages to build can be fine-tuned. Even the way the cross compiled software is installed on the target platforms is adjustable. Unlike OpenWrt, Openembedded can therefore be configured to implement the separation of configuration and software with an affordable effort.

Openembedded is very flexible and customisable. But due to its complexity, it is difficult to understand. Another barrier in Openembedded is the use of the bit-

bake tool for all compilation steps. The bitbake tool is not very well documented and rather complex. Bitbake interprets build recipes for cross compiling the particular packages. The syntax of this bitbake recipes is a mixture of Bourne Again SHell (Bash) and Python code. The main disadvantage of Openembedded is however the poor support for uClibc in the toolchain. In a test setup it was not possible to create a working uClibc cross toolchain. For all these reasons the author decided not to use Openembedded as cross compile environment for ADAM.

3.1.3 CLFS

Linux From Scratch (LFS) [19] is a project, which allows building a minimal Linux system entirely from the available sources. LFS provides therefore documentation, which contains step-by-step instructions to achieve the goal. These instructions are very detailed and every command line, which has to be executed is exactly explained. Additionally, the purpose of the used software packages and their initial configuration is described and alternatives for particular software packages are presented. This allows customising every aspect of the target system. The Cross Linux From Scratch (CLFS) project is a sub project of LFS, which provides such instructions for installing a cross toolchain and the necessary tools to build a minimal Linux system on a different architecture.

One major disadvantage of CLFS is that no automation of the build process is possible. Being a collection of documentation the LFS and CLFS projects provide no implementation of a build system. Nevertheless, the LFS and CLFS projects helped with useful information and hints during the planning and development of the ADAM build system. In addition, patches containing bug fixes for cross compilation of packages are provided by the LFS and CLFS projects, some of them are used also by ADAM.

3.2 Existing Configuration Frameworks

In this section three existing configuration frameworks for WMNs are analysed. Additionally, one monitoring solution for WMNs is presented.

3.2.1 MAYA

"A Tool For Wireless Mesh Networks Management (MAYA)" [20] is an implementation of a management solution for WMNs. MAYA is based on the OpenWrt firmware and the Adhoc On-Demand Vector Routing (AODV) [21] routing protocol. MAYA is an extension for OpenWrt, which provides mesh support and configuration of multiple nodes for the OpenWrt distribution. The mesh support is added by using an AODV agent on each node, which configures the routing.

MAYA uses the OpenWrt infrastructure installed on the nodes to configure each single node. In addition, MAYA provides mechanisms to change network configurations of multiple selected nodes. These changes are propagated through the

network by either sending a special User Datagram Protocol (UDP) message or by issuing a remote shell command. MAYA further provides a mechanism to add new nodes using their Internet Protocol Version 4 (IPv4) address. MAYA requires an additional central management server component, where the GUI for administration tasks is installed.

On one hand MAYA profits from the routing by AODV, new network configurations propagate faster than within the ADAM architecture, which distributes configurations periodically all two minutes and hop by hop. On the other hand MAYA depends on a working routing mechanism on the nodes which can also be regarded as a major disadvantage, if the routing mechanism should also be configurable.

In addition, static addresses have to be be configured on the nodes and the resulting network used for configuration is the same which the mesh clients are connected to. Moreover, MAYA can neither update software on the nodes automatically nor detect lost or misconfigured nodes.

3.2.2 ATMA

"A Framework for the Management of Large-Scale Wireless Network Testbeds (ATMA)" [22] is another implementation of a management solution for WMNs. Like MAYA, ATMA uses the OpenWrt firmware. ATMA is based on a client-server architecture. The server component is a central management server able to manage all mesh clients. The server beacons periodically its existence to the clients. This is achieved using the AODV routing protocol.

The mesh client is a software agent installed on the mesh nodes. This agent is written in the C programming language. The agent registers itself automatically to the server component with an IPv4 address from the automatic configuration range (169.254/16). Once registered to the server each mesh node is remotely manageable through the server.

ATMA supports four management tools. First the testbed configuration tool is used configure network parameters on a node in the testbed network. A second tool is the interference meter which measures channel interferences on all nodes. Third, the network monitoring component collects network statistics from all nodes. Monitoring is achieved by a modified version of Distributed Architecture for Monitoring Multi-hop Mobile Networks (DAMON) [23]. The fourth tool is the topology control tool, which allows simulating network topology changes without changing node locations.

Advantages of ATMA are the auto configuration of the mesh clients, the interference meter and the topology control tool. New clients can be easily added to the network. Like MAYA, ATMA requires the usage of a proper routing mechanism on the nodes and does not support software updates.

3.2.3 JANUS

"A Framework for Distributed Management of Wireless Mesh Networks (JANUS)" [24] is a management solution for WMNs implemented with the Java programming language. The JANUS architecture consists of four components. The first component is the mesh node itself, called the managed device. The mesh node runs the second component, the JANUS agent, a software process which listens on incoming connections. The third component is the mesh knowledge base which holds information about manageable parameters for the mesh node. The last component is the JANUS client, a software process running on the mesh nodes used for monitoring and controlling network parameters. The client polls the agent and replies on requests with the corresponding parameters found in mesh knowledge base. The JANUS architecture is very similar to the one used in Simple Network Management Protocol (SNMP).

JANUS is fully distributed and uses a peer-to-peer overlay network for communication. Mesh connectivity is provided by Mesh Connectivity Layer (MCL) [25]. Therefore, in the actual implementation JANUS is not platform independent. In addition, the JANUS implementation requires a Java virtual machine installed on the nodes.

3.2.4 DAMON

DAMON [23] is a monitoring solution for WMNs, which is implemented in a agent-sink architecture. The agents are running on the nodes and broadcast periodically their presence to the sinks. DAMON is a monitoring framework only, management of the mesh nodes is not supported. DAMON is used to collect network traffic statistics for later evaluation.

Like MAYA, DAMON uses the AODV routing protocol in the WMN. The DAMON agent is implemented in the Perl programming language and consists of collector modules, which send the monitored data to the sinks. The collected data consists mainly of AODV control messages and data traffic statistics. Like MAYA, DAMON is not functional without a properly set up routing mechanism on the nodes. Although being no configuration framework, DAMON is presented in this section due to its distributed architecture. In addition a modified version of DAMON is used in the ATMA implementation described in Subsection 3.2.2.

3.3 Discussion

None of the build systems presented fulfils all requirements for the automated build system needed by ADAM. None of the management solutions analysed is capable of software updates or service configuration on the nodes as required by ADAM. Moreover, the shown management solutions are not designed to be independent from the routing mechanism used in the WMN. In addition, the desired detection of misconfigured nodes or a safe update procedure for software images are not present

Feature	OpenWrt	Openembedded	CLFS
Build front-end	make	bitbake	none
Support for uClibc	by default	difficult	possible
Automated building	yes	yes	no
Documentation quality	usable	poor	only documentation
Package manager required	yes	no	no
Effort for integrating ADAM	high	acceptable	minimal

Table 3.1: Key features of existing build systems

Feature	MAYA	DAMON	ATMA	JANUS
Distributed architecture	yes	yes	yes	yes
All traffic encrypted	yes	n/a	n/a	n/a
Linux distribution used	OpenWrt	n/a	OpenWrt	n/a
Programming languages used	C, Shell	Perl	C	Java
Monitoring support	yes	yes	yes	yes
Support for configuration	yes	no	yes	yes
Support for software updates	no	no	no	no
Adhoc routing needed	yes	yes	yes	n/a
Adhoc routing used	AODV	AODV	AODV	n/a
Management server required	yes	yes (sinks)	yes	no

Table 3.2: Key features of existing monitoring/management solutions

in any implementation. For completeness, a comparison between the key features of the inspected build systems and management solutions is shown in Tables 3.1 and 3.2.

3.4 Assets and Drawbacks of SRM

SRM [16] introduced a configuration mechanism based on Cfengine [15] for WMNs built of PC Engines Alix and Wireless Router Application Platform (WRAP) nodes. Cfengine is a stable, secure and powerful framework for configuring Linux and UNIX-like systems in general, not only embedded devices. SRM showed that Cfengine is a modular and extensible solution for secure remote management of WMNs. Regarding embedded devices, Cfengine and its required libraries (OpenSSL, Berkeley DB) are relatively big binaries, SRM requires therefore node hardware which has at least 8 MByte permanent storage. On smaller systems no space for user defined software and tools would be left. SRM has showed some little drawbacks. SRM configuration modules are written in Bash .

SRM calculates reachable peers for Cfengine according the network settings of the nodes. Once a node is misconfigured somehow, its connection to the configuration system could be lost. Therefore a complicated and time consuming mechanism for detection of lost nodes and their reintegration in the network is used. Furthermore, SRM assumes a correct system time on the nodes. Therefore external time sources like Network Time Protocol (NTP) servers or battery driven real-time clocks are required. All modifications in network configurations have to be applied temporary first and checked for stability and reachability of nodes before being applied permanently. In order to make network configurations permanent the path from the management node to the reconfigured node has to be traversed at least three times, once for the propagation of the modifications, once for the answer that they are stable and temporary accepted, and the last time for the propagation of the message that the node can make these modifications permanent. This has a heavy impact on propagation times for network configuration changes and the overall convergence of the network. Moreover, during such network reconfigurations no other concurrent actions like for example updating firmware are possible.

External automatic address configuration systems like for example Dynamic Host Configuration Protocol (DHCP) are not fully supported in SRM, at least some static addresses have to be defined for a functional peer detection. If no static addresses are configured, no peers can be reached, as no addresses are known in the network. Moreover, there is neither Internet Protocol Version 6 (IPv6) support nor configuration of services like NTP or DHCP existing in SRM.

Chapter 4

ADAM Build System

The end product of the ADAM build system are appropriate software images for all supported embedded devices. As described in Chapter 2, a cross toolchain is required for compiling software for particular embedded devices. The ADAM build system automates the installation of an appropriate cross toolchain as well as the cross compilation of the selected software for the target board. The ADAM build system therefore requires only the source code of the used software, and is therefore hardware independent. The ADAM build system consists of two main parts, the software build process and the image creation process. The first part, the software build process simplifies the cross compilation of software for different wireless mesh nodes. In this chapter the internal functionality and structure of the software build process is presented. The installation of the ADAM build system, its setup procedures for building for a specific platform, the toolchain building and the usage of the toolchain to cross compile software for the target platform are shown.

As second part of the ADAM build system, the image creation process, which generates appropriate software and configuration images for the mesh nodes out of the cross compiled software is described in detail in Chapter 5. The ADAM build system requires a recent Linux workstation with accurate performance and root access. Nothing else is needed, the ADAM build system creates the required software images for supported node devices with a few command line invocations. The ADAM build system is extensible and customisable. Support for new software packages or new target platforms is added with a small effort.

Once a board is supported by the ADAM build system, it can participate in a WMN managed by the ADAM configuration framework described in Chapter 6.

4.1 Prerequisites for the Host Platform

The requirements and procedures for building a toolchain, which cross compiles software packages for mesh nodes were described in a general way in Chapter 2. Two hardware platforms are involved in the build process. The first platform, the host platform, actually does all computation intensive work from generating a

toolchain via cross compiling to image generation. The second platform, the target platform, is the embedded node device for which the software is cross compiled. The host platform should be selected by the developer in terms of accurate performance and storage capacity. The prerequisites concerning required tools and operating system versions for the host system should be as few as possible, which allows using a wide variety of host platforms.

Some constraints concerning the host platform are however unavoidable. It should run a recent Linux kernel (version 2.6.22 or higher), older versions were not tested and therefore not officially supported. The ADAM build system does not require a special Linux distribution, generally all distributions are supported, but some development tools and programs are needed by the build system. For example at least a C compiler has to be installed, it is required to create the intermediate cross compiler. Other necessary tools and programs have to be installed on the host system. Table 4.1 shows all these required packages, their minimal version needed and the functions provided by them. These tools can be installed regularly with the package manager of the particular distribution. In addition, the development parts associated to a particular package have to be installed if the distribution uses such development packages (Debian, Ubuntu, Fedora). The ADAM build system checks for the existence of all required tools and their minimal versions in the setup procedure for a specific platform.

4.2 Organisation and Structure

The software build process for a particular embedded system is a complex procedure and requires many dependent tasks to be executed in the correct order. The goal of the build process is to cross compile software for the embedded target device using only the available source code. The ADAM build system tries to simplify this process by providing an easy understandable and extensible infrastructure for configuration and execution of these tasks.

The ADAM build system is therefore organised in three main parts, one command line build front-end, called *build-tool*, and the directories *buildconfig* and *buildscripts*, containing all build task configuration data, and the package build scripts used to execute the different tasks, respectively.

4.2.1 Build-Tool Front-End

The command line front-end of the ADAM build system is *build-tool*. It is used to control the whole software build process, from source code to cross compiled software for a specific target board. The command line front-end *build-tool* is therefore the only tool needed by the end user for completing the software build process, if no further customisation is needed. Such customisation actions for the ADAM build system are explained in Section 4.4.

Package	Minimal version	Provided programs
Bash	2.05a	Shell used by the build system
Binutils	2.12	ELF binary tools
Bison	2.3	Parser generator
Bzip2	1.0.2	Bzip2 compression utilities
Coreutils	5.0	Programs for managing the basic system properties
Diffutils	2.8	Utilities that show the differences between files
Findutils	4.1.20	Programs to find files or directories
Flex	2.5.33	Tool that creates pattern recognition programs
Gawk	3.0	Programs for manipulating text files
GCC	2.95.3	GNU Compiler Collection
Glibc	2.2.5	GNU C library
Grep	2.5	Tools for searching through files
Gzip	1.2.4	Gzip compression utilities
Make	3.79.1	Program for compiling source code
Patch	2.5.4	Tool for modifying or creating files
Sed	3.0.2	Stream editor
Tar	1.14	Tar archive utilities
Curl	7.18	Program for downloading files
Texinfo	4.4	Info documentation utilities
Pkg-config	0.22	Tool for reading package compile information

Table 4.1: Required packages for the host platform

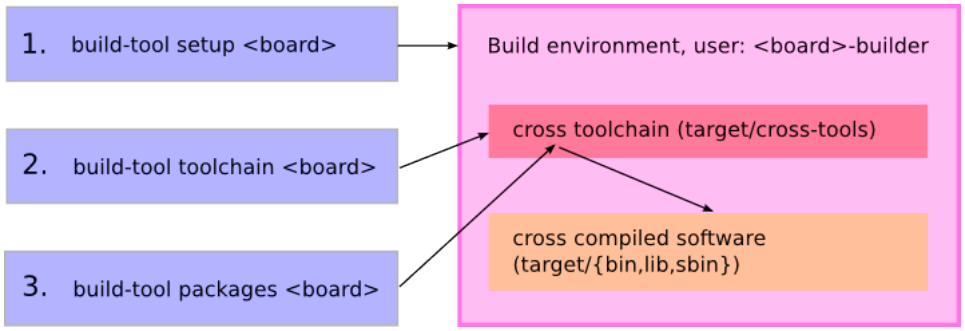


Figure 4.1: The three *build-tool* main modes

If someone wants to build the already supported packages for an already supported target platform, only three different invocations of *build-tool* on the command line are required to achieve this goal and complete the software build process. These three main modes of *build-tool* include the following procedures which are also shown in Figure 4.1. Each procedure requires the successful completion of the preceding procedure. Listing 4.1 shows the different command line arguments, which are used to control the three main modes of *build-tool*.

1. *build-tool setup <board>*

   The setup procedure for a specific target board includes a sanity check of the host system, adds a build user, sets up its environment and collects the right build configuration data and copies it in the build users home directory. The build user is mainly used to provide a clean build environment.

2. *build-tool toolchain <board>*

   The build user and the build configuration data in its home directory are used to compile and install the cross toolchain packages.

3. *build-tool packages <board>*

   Use the cross toolchain installed for the specified target board to cross compile and install all the needed software packages for the target board.

```
root@monk:~/image-builder# ./build-tool
Usage ./build-tool setup     <boardname>
      ./build-tool toolchain <boardname> [package1 package2...]
      ./build-tool packages  <boardname> [package1 package2...]

root@monk:~/image-builder#
```

Listing 4.1: Usage of the *build-tool* front-end

4.2.2 Build Users

For each supported target board the ADAM build system utilises a unique user id on the host system for compilation and installation of the software. Such a user is called build user and corresponds to a normal user id on the host system. A build user for a specific target follows a naming convention and is called target-builder. Due to access permissions, each build user is able to compile and install software only in its home directory. All processes used for compilation and installation run with the unprivileged user id of the build user.

This has the advantage, that an eventually bad written package build script is not able to overwrite critical system files on the host system. It further enables the possibility of multiple simultaneous build processes for different target platforms on one host system.

But using build users has the disadvantage of a more complex setup procedure. Build users and their home directories have to be created in this setup procedure for a specific target platform. Normally adding users requires root privileges and therefore *build-tool* requires root rights on the host system at least for this task.

4.2.3 Configuration of the Build Process

The build process for a particular platform is configured with several configuration files in plain text format. First of all, each supported target platform is described with a corresponding build profile. It is the most important configuration file and contains mandatory information about the processor and architecture of the target board. These build profiles are described in more detail in Subsection 4.2.4. Second, in addition to the build profile, a platform dependent Linux kernel and uClibc configuration file are required for each supported target board. These two configuration files are required to compile a platform dependent Linux kernel and uClibc library. Third, configuration files can also be patches for a specific software package. Patches are text files containing user defined changes and fixes for package source code, mostly in a file format provided by the diff program. All these configuration files used for steering and fine tuning of the build process are contained in the *buildconfig* directory. The *buildconfig* directory therefore contains required information about the target board, configuration files for particular packages, and patches for the different software packages. It is organised in several files and directories, a detailed view of them is also shown in Figure 4.2.

The special directory *_generic* contains build task configuration files common to all target platforms. In addition, each supported target platform needs a corresponding directory containing platform specific configuration files, which are either not already contained in the *_generic* directory, or which overwrite build configuration files contained in the *_generic* directory.

This structure behaves like a simple inheritance chain. As long as a specific build configuration file is not present in the specific directory for the target plat-

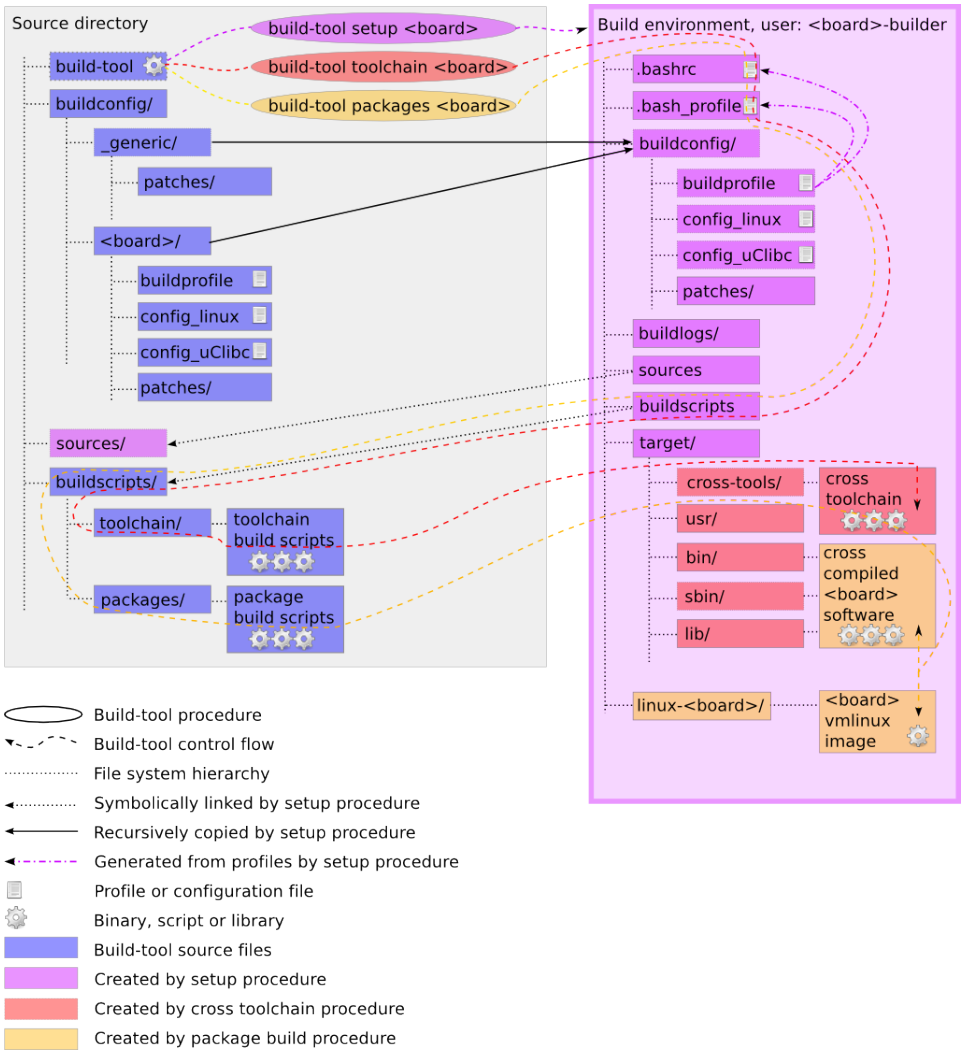


Figure 4.2: Details of the software build process

form, the corresponding file in the *_generic* directory is used. Practically this is achieved by copying the contents of *_generic* to *buildconfig* in the home directory of the build user in a first step. By copying the contents of the target specific directory in a second step, certain *_generic* build configuration data is overwritten selectively for the target platform used.

Patches for software packages are located in the *patches* sub-directory of each build configuration directory and ordered according the package name and its version. Therefore if a patch has to be applied for all platforms, it has to be contained in the *_generic/patches* sub-directory. If a patch has only to be applied for a particular platform it has to be located inside the platform specific *<board>/patches* sub-directory. Patches are applied to unpacked source code archives with the patch tool. The ADAM build system provides an internally defined function which collects and applies patches automatically if the version and the name of the package matches. With this structure new patch files for a particular package version and target platform have to be copied only to the right *patches* directory to be automatically applied by the build system.

4.2.4 Build Profiles

As seen in the previous subsection, all attributes of a specific target platform are contained in a special configuration file. This configuration file is the build profile for the target platform and mandatory for all supported platforms. A build profile has to be named *buildprofile* and is located in the target specific directory inside the *buildconfig* directory. The format of the file is plain text and contains assignments in the form *VARIABLENAME=value*. Required variables are the name of the target board, its architecture and processor type and the default selection of software packages to build for the target board. In addition, a shell function has to be defined containing the commands to create a bootable kernel image from the generic kernel image compiled for the platform. Listing 4.2 shows the build profile for the Meraki [5] board containing the different required parameters.

```
# This is the build profile for the Meraki board

# Name of your board
BOARDNAME=meraki
# Target triplet
BOARDTARGET=mips-linux-uclibc
# Architecture
BOARDARCH=mips
# If BOARDARCH=mips, you need to define MIPSLEVEL.
MIPSLEVEL=1

# These packages get built by default:
BOARDPACKAGES="zlib openssl curl dropbear openntpd nostromo busybox linux
    madwifi wireless-tools module-init-tools iproute2 iputils ipv6calc
    iptables flex radvd sudo db cfengine olsrd"

# The board-specific function ${BOARDNAME}-genimage.
# For meraki this is pretty easy, just gzip and copy
```

```
# ${KERNELDIR}/arch/${BOARDARCH}/boot/vmlinux.bin.
meraki-genimage()
{
    cp ${KERNELDIR}/arch/${BOARDARCH}/boot/vmlinux.bin ${BOARDNAME}-image-
        ${1}.bin
    gzip -f -9 ${BOARDNAME}-image-${1}.bin
    chmod 644 ${BOARDNAME}-image-${1}.bin.gz
}
```

Listing 4.2: Build profile for the Meraki board (*buildprofile*)

4.2.5 Build Environment

The ADAM build system uses the shell environment variables of each build user to create a so called build environment dedicated to a specific target board. A build environment for a particular target board is created by exporting shell environment variables to the build users shell. These shell environment variables define the required parameters used for the build process like the different directories used, architecture and processor type related information and the default package set which is built for the given target platform. These environment variables are set up by the normal Bash initialisation profiles *.bash_profile* and *.bashrc* located in the build users home directory. The initialisation profiles have to be generated by the setup procedure from the information found in the build profile for the the selected target board. Listings 4.3 and 4.4 show examples of both Bash initialisation profiles for the meraki-builder build user.

```
set +h
umask 022
unset CFLAGS
unset CXXFLAGS
export LC_ALL=POSIX
export BUILDDIR="${HOME}"
export BUILDCONFDIR="${BUILDDIR}/buildconfig"
export INSTALLDIR="${BUILDDIR}/target"
export SRCDIR="${BUILDDIR}/sources"
export PATH=${INSTALLDIR}/cross-tools/bin:/bin:/usr/bin
export CROSS_TARGET=mips-linux-uclibc
export BOARDNAME=meraki
export BOARDTARGET=mips-linux-uclibc
export BOARDARCH=mips
export BOARDPACKAGES="zlib openssl curl dropbear openntpd nostromo busybox
     linux madwifi wireless-tools module-init-tools iproute2 iputils
    ipv6calc iptables flex radvd sudo db cfengine olsrd"
export MIPSLEVEL=1
export CC="mips-linux-uclibc-gcc"
export CXX="mips-linux-uclibc-g++"
export AR="mips-linux-uclibc-ar"
export AS="mips-linux-uclibc-as"
export LD="mips-linux-uclibc-ld"
export RANLIB="mips-linux-uclibc-ranlib"
export READELF="mips-linux-uclibc-readelf"
export STRIP="mips-linux-uclibc-strip"
export BUILD="-mabi=32"
```

Listing 4.3: Bash initialisation profile (*.bashrc*)

```
exec env -i HOME=${HOME} TERM=${TERM} PS1='buildenv:\u:[\w ] $ ' /bin/bash
```

Listing 4.4: Bash initialisation profile (*.bash_profile*)

4.2.6 Package Build Scripts

A package build script is a Bash shell script with the *.sh* file extension. Every single task which has to be executed in the build process is described with a corresponding package build script in the *buildscripts* directory (e.g. *package.sh*). Package build scripts contain the shell commands for downloading, compiling and installing software packages.

Package build scripts are split up into two categories, toolchain build scripts and package build scripts. Toolchain build scripts are used for building cross toolchain related packages and are located in the *toolchain* sub-directory. Normal build scripts are used for building all other software packages and are contained in the *packages* sub-directory.

The first section of a package build script is used for the definition required variables, like the version of the package, its download location and a check sum for its source archive. In addition, some commonly used functions for downloading and patching the source code are loaded in this section. The second section is the current task to be performed. The build task for an average package is generally divided into the following six finer steps.

1. Download of package source archive

   Fetch the package source archive from its download location and verify its integrity by calculating and comparing the check sum. This sub-task can be done by the internally defined functions *download_gz()*, *download_bz2()* or *download_file()* for gzip or bzip2 compressed tar archives and other file formats respectively.

2. Decompression of package source archive

   Unpack the source archive and change into the created source directory. Normally two shell commands are enough to achieve this.

3. Patching package source code

   Apply all matching patches found for this package version in the *buildconfig* directory. This sub-task can be done by the internally defined function *patch_src()*.

4. Package configuration

   Configure the package for cross compiling. This sub-task differs from package to package. It can be very complex and consist of several shell commands or it may be nothing at all.

5. Package compilation

   Compile the package. Mostly one more or less complex make command solves this issue.

6. Package installation

   Install the package to the right place. This is generally a problem of copying the right files to the correct destination, which is not as easy as it seems for some packages. This installation sub-task must consider the image creation process described in Chapter 5. Cross compiled binaries and libraries have to be installed to the home directory of the build user inside the *target* directory, while configuration files and init scripts have to be installed to the configuration templates located in the *config* directory.

These sub-tasks can be individually defined or omitted in the corresponding package build script. There even exist package build scripts which do not belong to a particular package, they are used to execute tasks which are not related to any package, like e.g. cleaning up the build environment after the installation of the cross toolchain.

An example for a normal package build script is shown in Listing 4.5. It is the build script for the cross compilation and installation of the commonly used zlib compression library. The name of the build script is derived from the name of the package, e.g. *zlib.sh*. Another example for a package build script is shown in Listing 4.6. It is a toolchain build script called *cross-gcc.sh*, which compiles and installs the final cross compiler during the cross toolchain installation process. As already seen, it is possible that a build script does not compile software, but performs other important tasks in the build process. An example for this kind of a build script is shown in Listing 4.7. It is responsible for cleaning up the build environment after the cross toolchain installation. Since no package is related to this build script its file name except for its extension can be chosen freely.

```
#!/bin/bash

################################################
. ${BUILDDIR}/buildscripts/functions

VERSION="1.2.3"
SHA1SUM="967e280f284d02284b0cd8872a8e2e04bfdc7283"
URL="http://www.zlib.net"
FALLBACK="http://danielbalsiger.ch/image-builder"
BUILD_DEPS="toolchain"
################################################

download_bz2 zlib &&

cd ${BUILDDIR} &&
tar -xjvf ${SRCDIR}/zlib-${VERSION}.tar.bz2 &&
cd zlib-${VERSION} &&

patch_src zlib &&
```

```
sed -e 's/-O3/-Os/g' -i configure &&
CC="${CC} -Os -fPIC" ./configure --prefix=/usr --shared &&
make &&

make DESTDIR=${INSTALLDIR} install &&
mv -v ${INSTALLDIR}/usr/lib/libz.so.* ${INSTALLDIR}/lib &&
ln -svf ../../lib/libz.so.1 ${INSTALLDIR}/usr/lib/libz.so &&
ln -svf libz.so.1 ${INSTALLDIR}/lib/libz.so &&

cd ${BUILDDIR} &&
rm -rf zlib-${VERSION}
```

Listing 4.5: Example of a package build script (*zlib.sh*)

```
#!/bin/bash

################################################
. ${BUILDDIR}/buildscripts/functions

VERSION="4.2.4"
SHA1SUM="bb20efc7750fe0d6172c5945572bf036fe59d3dd"
# this is 4.1.2 SHA1SUM="7981b8d1b58b10ddfd7d5142eab16352d9206f3b"
URL="http://ftp.gnu.org/gnu/gcc/gcc-${VERSION}"
FALLBACK="http://danielbalsiger.ch/image-builder"
################################################

download_bz2 gcc &&

cd ${BUILDDIR} &&
tar -xjvf ${SRCDIR}/gcc-${VERSION}.tar.bz2 &&
cd gcc-${VERSION} &&

patch_src gcc &&

mkdir -v ../gcc-build &&
cd ../gcc-build &&
../gcc-${VERSION}/configure --prefix=${INSTALLDIR}/cross-tools --target=${
    CROSS_TARGET} --disable-multilib --with-sysroot=${INSTALLDIR} --
    disable-nls --enable-shared --enable-languages=c,c++ --enable-
    __cxa_atexit --enable-c99 --enable-long-long --enable-threads=posix --
    enable-libssp &&
make &&
make install &&

cd ${BUILDDIR} &&
rm -rf gcc-${VERSION} gcc-build
```

Listing 4.6: Example of a toolchain build script (*cross-gcc.sh*)

```
#!/bin/bash

################################################
# No download, no source
################################################

echo export CC=\""${CROSS_TARGET}-gcc\"" >> ~/.bashrc &&
echo export CXX=\""${CROSS_TARGET}-g++\"" >> ~/.bashrc &&
echo export AR=\""${CROSS_TARGET}-ar\"" >> ~/.bashrc &&
echo export AS=\""${CROSS_TARGET}-as\"" >> ~/.bashrc &&
echo export LD=\""${CROSS_TARGET}-ld\"" >> ~/.bashrc &&
```

```
echo export RANLIB=\""${CROSS_TARGET}-ranlib\"" >> ~/.bashrc &&
echo export READELF=\""${CROSS_TARGET}-readelf\"" >> ~/.bashrc &&
echo export STRIP=\""${CROSS_TARGET}-strip\"" >> ~/.bashrc &&
if [ "${BOARDARCH}" == "mips" ] ; then
    echo export BUILD=\""-mabi=32\"" >> ~/.bashrc
fi &&
cd ${BUILDDIR} &&
tar -cjvf toolchain-${BOARDNAME}.tar.bz2 target &&
touch ${BUILDDIR}/.toolchain &&

cd ${BUILDDIR}
```

Listing 4.7: Example of a task build script (*cleanup.sh*)

4.3 Building for a Specific Target Platform

In the previous sections the organisation and structure of the build system were outlined. This section explains in more detail how this structure is used to cross compile target board software with *build-tool* in three simple steps. The roles of the build user, its build environment and the build profile for the target board were already explained. If such a build profile is available, the build system knows everything to start the build process for a particular target board. Figure 4.2 shows the control flow of the three main operations of *build-tool* and the involved files and directories.

4.3.1 Target Board Setup Procedure

Before compiling any software or installing a cross toolchain, the host system for the selected target platform has to be set up. The target board setup procedure, also shown in Figure 4.2, selects a supported target board and sets up all build parameters for the chosen platform. First of all the setup procedure does the sanity checks of the host system as already mentioned in Section 4.1, adds the build user and creates its home directory, if neither the build user nor its home directory are existing. In a second step the build environment for the build user is created and some necessary directories are made in its home directory, namely *target*, where the cross compiled software will be installed and *buildlogs*, where build logs of each package are saved. Additionally the setup procedure does create two symbolic links in the build users home directory to the real *buildscripts* and *sources* directories, located in the source directory, shown also in Figure 4.2. Therefore the build scripts and downloaded source archives are shared among all build users, which enables to reuse already downloaded source archives for other platforms. On the contrary, each build user gets its own copy of the *buildconfig* directory for profiting of the inheritance chain already explained in Subsection 4.2.3. The normal setup procedure for a particular target board is simply performed by executing *build-tool* with the mandatory setup and target board arguments, as shown in Listing 4.8.

4.3.2 Cross Toolchain Installation

After a successful setup procedure, the build user has been created and its build environment is set up to compile software for the selected target board. Now the cross toolchain can be installed, which is achieved with the special toolchain build scripts described in Subsection 4.2.6. The default toolchain packages and their installation order is hard-coded, as it is seldom necessary to change neither the used build scripts nor their order. Nevertheless, the toolchain building behaviour can be easily adapted. Subsection 4.4.1 outlines this process. The normal procedure for a toolchain installation is started by *build-tool* with the mandatory toolchain and target board arguments, as shown in Listing 4.9. The toolchain is installed to the *target/cross-tools* directory located in the build users home directory.

4.3.3 Building Software Packages

After a successful cross toolchain installation, the cross compiler for the target platform is available in the *target/cross-tools* directory. It is used to cross compile software packages for the target platform. All these software packages are built and installed with the help of the package build scripts described in Subsection 4.2.6. Each package has its corresponding package build script responsible for cross compiling and installing its associated binaries and libraries. As seen before the default package set to build and its order is defined in the build profile of the target board. Each package build script installs its cross compiled binaries to *target/sbin* and *target/bin*, and shared libraries to *target/lib* inside the build users home directory, where they can be found by the image creation process described in Section 5.2. All other directories in *target* are not used by the image creation process, especially the *target/usr* directory. Therefore development related files, for example compile information used by the pkg-config program, static libraries, include files and all documentation should be installed to the *target/usr* directory. Software packages are built and installed by executing *build-tool* with the mandatory packages and target board arguments, as shown in Listing 4.10.

4.3.4 Configuration Files and Initialisation Scripts

If a package requires special configuration files or init scripts, in short files which are not cross compiled binaries or libraries, these files have to end up in the configuration images described in Subsection 5.1.2 and therefore have to be treated specially. Unlike cross compiled software they must not be installed to the *target* directory inside the build users home directory, as they are not part of the software images created out of these directories. Package build scripts do not have to perform any action on such configuration files or init scripts. Their installation can be simply omitted in the installation sub-task of the package build script, they are only treated in the configuration image creation process described in Section 5.3. However, it is advisable to check for their existence and correctness in the config-

uration templates located in *config*, when packages are upgraded from one version to another.

4.3.5 Sample Output of Build-Tool

This section provides some sample output of *build-tool* during the setup, the cross toolchain installation and the package installation procedures for the Meraki target board.

```
root@monk:~/image-builder# ./build-tool setup meraki
Your host system should have at least these programs installed,
in the following minimal versions:

      Bash-2.05a
      Binutils-2.12
      Bison-2.3
      Bzip2-1.0.2
      Coreutils-5.0
      Diffutils-2.8
      Findutils-4.1.20
      Flex-2.5.33
      Gawk-3.0
      Gcc-2.95.3
      Glibc-2.2.5
      Grep-2.5
      Gzip-1.2.4
      Make-3.79.1
      Patch-2.5.4
      Sed-3.0.2
      Tar-1.14
      Curl (for downloads, any version)
      Texinfo 4.4
      Pkg-config 0.9

I will search for them now and print their version:

GNU bash, version 3.2.39(1)-release (i486-pc-linux-gnu)
GNU ld (GNU Binutils for Ubuntu) 2.18.93.20081009
bison (GNU Bison) 2.3
bzip2, a block-sorting file compressor.  Version 1.0.5, 10-Dec-2007.
chown (GNU coreutils) 6.10
diff (GNU diffutils) 2.8.1
find (GNU findutils) 4.4.0
flex 2.5.35
GNU Awk 3.1.6
gcc (Ubuntu 4.3.2-1ubuntu12) 4.3.2
GNU C Library development release version 2.8.90, by Roland McGrath et al.
GNU grep 2.5.3
gzip 1.3.12
GNU Make 3.81
patch 2.5.9
GNU sed version 4.1.5
tar (GNU tar) 1.20
curl 7.18.2 (i486-pc-linux-gnu) libcurl/7.18.2 OpenSSL/0.9.8g zlib/1.2.3.3
     libidn/1.8
makeinfo (GNU texinfo) 4.11
pkg-config 0.22

If your package version requirements are fulfilled,
```

```
you may continue with the installation.
Otherwise install the missing tools or choose another host system.
You have been warned!

Press 'y' to continue installation, 'n' to abort
y

Checking for build user


Adding user and group meraki-builder:


Setup done for building a meraki toolchain.

root@monk:~/image-builder#
```

Listing 4.8: Output of the target board setup procedure

```
root@monk:~/image-builder# ./build-tool toolchain meraki

Toolchain mode detected, I will install the following packages in order
    from left to right:

base-files linux-headers cross-binutils uclibc-headers cross-gcc-static
    uclibc cross-gcc cleanup

Packages will be built in:     /home/meraki-builder
Package sources are in:        /home/meraki-builder/sources
Extra configuration is in:     /home/meraki-builder/buildconfig
Packages will be installed to: /home/meraki-builder/target
Packages get compiled by:      /home/meraki-builder/buildscripts/toolchain
    /<package>.sh
Buildlog is in:                /home/meraki-builder/buildlogs/<package>.
    buildlog
Compiler used:                 gcc
Target system triplet is:      mips-linux-uclibc

Are these values reasonable ? Begin installation [y/n]
y

Sat Mar 28 09:03:43 CET 2009: Installing package base-files...
Sat Mar 28 09:03:43 CET 2009: Installing package linux-headers...
Sat Mar 28 09:04:45 CET 2009: Installing package cross-binutils...
Sat Mar 28 09:08:46 CET 2009: Installing package uclibc-headers...
Sat Mar 28 09:08:54 CET 2009: Installing package cross-gcc-static...
Sat Mar 28 09:16:12 CET 2009: Installing package uclibc...
Sat Mar 28 09:17:31 CET 2009: Installing package cross-gcc...
Sat Mar 28 09:28:56 CET 2009: Installing package cleanup...

Sat Mar 28 09:29:35 CET 2009: Toolchain installation of packages base-
    files linux-headers cross-binutils uclibc-headers cross-gcc-static
    uclibc cross-gcc cleanup completed.

root@monk:~/image-builder#
```

Listing 4.9: Output of the cross toolchain installation procedure

```
root@monk:~/image-builder# ./build-tool packages meraki

This will install the following packages in order from left to right:

zlib openssl curl dropbear openntpd nostromo busybox linux madwifi
    wireless-tools module-init-tools iproute2 iputils ipv6calc iptables
    flex radvd sudo db cfengine olsrd

Packages will be built in:     /home/meraki-builder
Package sources are in:        /home/meraki-builder/sources
Extra configuration is in:     /home/meraki-builder/buildconfig
Packages will be installed to: /home/meraki-builder/target
Packages get compiled by:      /home/meraki-builder/buildscripts/packages
    /<package>.sh
Buildlog is in:                /home/meraki-builder/buildlogs/<package>.
    buildlog
Cross-Compiler used:           mips-linux-uclibc-gcc
Target system triplet is:      mips-linux-uclibc

Are these values reasonable ? Begin installation [y/n]
y

Sat Mar 28 09:34:07 CET 2009: Installing package zlib...
Sat Mar 28 09:34:10 CET 2009: Installing package openssl...
Sat Mar 28 09:36:52 CET 2009: Installing package curl...
Sat Mar 28 09:37:57 CET 2009: Installing package dropbear...
Sat Mar 28 09:38:38 CET 2009: Installing package openntpd...
Sat Mar 28 09:38:52 CET 2009: Installing package nostromo...
Sat Mar 28 09:38:55 CET 2009: Installing package busybox...
Sat Mar 28 09:40:01 CET 2009: Installing package linux...
Sat Mar 28 09:46:49 CET 2009: Installing package madwifi...
Sat Mar 28 09:47:17 CET 2009: Installing package wireless-tools...
Sat Mar 28 09:47:20 CET 2009: Installing package module-init-tools...
Sat Mar 28 09:47:26 CET 2009: Installing package iproute2...
Sat Mar 28 09:47:49 CET 2009: Installing package iputils...
Sat Mar 28 09:47:52 CET 2009: Installing package ipv6calc...
Sat Mar 28 09:47:59 CET 2009: Installing package iptables...
Sat Mar 28 09:48:13 CET 2009: Installing package flex...
Sat Mar 28 09:48:31 CET 2009: Installing package radvd...
Sat Mar 28 09:48:40 CET 2009: Installing package sudo...
Sat Mar 28 09:48:53 CET 2009: Installing package db...
Sat Mar 28 09:50:09 CET 2009: Installing package cfengine...
Sat Mar 28 09:51:20 CET 2009: Installing package olsrd...

Sat Mar 28 09:51:31 CET 2009: Installation of packages zlib openssl curl
    dropbear openntpd nostromo busybox linux madwifi wireless-tools module
    -init-tools iproute2 iputils ipv6calc iptables flex radvd sudo db
    cfengine olsrd completed.

root@monk:~/image-builder#
```

Listing 4.10: Output of the package installation procedure

4.4 Customisation of the Build System

The ADAM build system follows a modular and extensible approach. It has been developed to be as extensible and configurable as possible, every component can

be diversified by the developer according to future needs. Due to its modular design, practically the whole behaviour of the build procedure can be adjusted. Such adjustments can be simple tasks as changing the default package selection for a specific board or the order these packages are built, and more complex tasks, like adding support for a new package or even a new target platform. This section outlines the different customisation options of the ADAM build system.

4.4.1 User Defined Package Selection

Each build profile contains the default package selection to be built for a specific target board. An example is shown in Listing 4.2. The default package selection can be modified by editing the build profile and adjusting the *BOARDPACKAGES* variable which defines the default package list. The build order of packages is defined by the order of the appearance of the package in the default package list.

However, it is also possible to provide a temporary list of packages including their build order as command line argument to *build-tool*. With this technique it is possible to build any package set in a requested order without needing to change the default package list defined in the build profile for the board. If for example packages foo and bar have to be built and installed in this order for the board target platform , the command line to be executed looks as follows.

build-tool packages board foo bar.

To cross compile and install only a single package, the command line provided list can also contain only one single package. If the command line provided list is empty the default package list defined in the build profile is taken. With the same approach it is possible to change the internally hard-coded set of packages which is used in the cross toolchain mode of *build-tool*.

4.4.2 Adding Support for New Packages

Adding support for a new package in the build system is another customisation option often required by the developer. Therefore only a new package build script has to be written.

If a new package foo has to be supported, the developer needs to create a new package build script named *foo.sh*, which has to be located in the *buildscripts/packages* directory. This build script itself has to contain all commands to download, unpack, patch, configure, cross compile and install the package foo.

These commands have to be defined by the developer, and should do the tasks described in Subsection 4.2.6 and install all files belonging to the package to the locations explained in Subsection 4.3.3. Existing build scripts and a special skeleton build script can provide a basis for writing new build scripts.

Once the build script for the new package has been written, the new package can be built for any platform supported by the build system. Any new package build script should be tested on any supported target platform in order to guarantee proper functionality of the build system. Sometimes it is even required to add

additional platform dependent patches for the package to ensure a proper compilation on all supported target platforms. In addition, if the package foo uses any configuration files or special init scripts, the developer should add all these files to the configuration templates described in Section 5.3.

4.4.3 Adding Support for New Target Platforms

Adding support for new target boards is the most work intensive customisation action for the build system. In order to support new target boards, the following three steps are necessary: Creation of a Linux kernel configuration file, a uClibc configuration file and a build profile.

First of all, a Linux kernel configuration file for the new target board has to be provided by the developer. Due to the high degree of hardware dependency of the Linux kernel, this configuration file cannot be generated by the build system. However, the Linux kernel configuration files for already supported platforms can be taken as template to create the new configuration file more easily.

For the creation of a functional toolchain for the new target board, a configuration file for uClibc must be provided. Depending on the architecture of the new target board this can be a more or less work intensive task. If a board with the same processor architecture, e.g. i386 or Microprocessor without Interlocked Pipeline Stages (MIPS) , is already supported by the build system, the existing uClibc configuration file for the already supported board can be taken for the new board. Otherwise the uClibc configuration file has to be created from scratch by the developer. Like in the case of kernel configuration files, already existing uClibc configuration files for other architectures can provide a basis for the new configuration file. The Linux kernel as well as the uClibc build procedures provide a menu based configuration mechanism to create these files.

Since each supported target board requires a build profile, such a build profile has to be created in a third step. This is no difficult task in general, since only a few variables have to be defined in the build profile. Once a Linux kernel and uClibc configuration file as well as the build profile for the target board have been created, the build system should be able to build a toolchain and all other packages for the new target platform.

However, a complete test with all available packages should be performed to validate the build system for the new target board. Maybe some package build scripts have to be extended to work for the newly defined target board, and additional patches have to be provided to compile a package successfully.

Chapter 5

ADAM Images

The ADAM build system uses different images to deploy software as well as configuration data to the nodes. This chapter will explain the differences between the images used, their purpose, their creation process and their installation to different node platforms. Moreover, the boot process of a deployed node is outlined.

5.1 Image Types

Several image types are required for ADAM. A node in an ADAM network uses a configuration and a software image.

Every node, which is member of a network managed by the ADAM configuration framework needs configuration information like network settings and keys for a secure communication with other nodes. This configuration information is vital for a working configuration framework and differs naturally from one node to another. Further, each node platform should use exactly the same cross compiled software. For this purpose ADAM provides the two image types, software images and configuration images.

Software images contain all cross compiled software common to a specific node platform, while configuration images contain node specific configuration data which differs from node to node. Therefore one software image per node type and one configuration image for each particular node has to be provided by the image generation process. The split between node configuration and node software allows updating software images without influencing the configuration of a particular node. Without this split it would be necessary to embed all node specific configuration information directly into the software image and lead to a unique software image for each node. In this case every image for a single node has to be distributed inside the network. Software updates would be much more complicated.

An individual configuration image has to be installed on each node to use a software image. Without a configuration image a software image is not functional, in fact the node does not even boot properly without it. Since each node has its own configuration image, there have to be as much configuration images as nodes

in the network. Different software images are only needed if different node types are used, exactly one software image per node type has to be provided.

The size of a software image depends on the target board and the amount of software packages used. Software images which contain the default package selection have a size of about 5 MByte (e.g. 5.2 MByte on Alix [6], 5.5 MByte on Meraki [5] nodes). The size of a configuration image is 4 MByte for all node types. However, the default configuration files stored in the configuration image use less space (e.g. 400 KByte). On nodes with small storage footprints, it is even required to install only the contents of the configuration image to the flash partition, as the configuration image itself would not fit there.

5.1.1 Software Images

Software images contain all the cross compiled software for a specific node platform, including all user space programs, libraries and the Linux kernel. The cross compiled user space programs and libraries are contained in a compressed initramfs cpio archive added to the Linux kernel image at compile time. The resulting software image is a bootable Linux kernel image for the node platform. The format of the software image therefore differs for each node platform, it depends on the boot loader used, which has to be able to read the image to boot correctly. At boot time the initramfs archive embedded in the software image is mounted read-writable as root file system by the Linux kernel. All content of the initramfs archive is loaded into RAM and therefore modifications are not permanent over reboots.

Initramfs archives are provided in a compressed format. This has the advantage, that the uncompressed root file system in RAM is bigger than the corresponding compressed embedded initramfs archive on secondary storage. On nodes with a small secondary storage capacity like the Meraki, this technique allows using root file systems whose uncompressed content would not even fit on the available secondary storage.

If new packages have to be added or existing packages have to be replaced in a software image, the generation and installation of a whole new software image is required, which is a disadvantage of all-in-one software images compared to a package manager based solution, where only the affected package would have been added or replaced. Concerning space issues on the contrary, the all-in-one software image approach is preferable, more software can be installed with it. Therefore ADAM software images use exactly this all-in-one image approach.

5.1.2 Configuration Images

Since the contents of software images cannot be modified individually without installing a new software image, a node needs a possibility to save individual files on secondary storage. For this purpose the configuration image is used. A configuration image contains all individual node specific configuration files and can be used to save in principle any file on secondary storage as long as enough space is

left there. The contents of a configuration image are made available to a node by mounting them to a special mount point */mnt/config*, and copying them to the root file system during the boot process. Which files are contained in a configuration image is defined in the configuration file */etc/configfiles* found on the nodes. With the help of this file, the contents of a configuration image can be dynamically adjusted at run time. The init script used at boot time to load the initial contents is as well used to store and load configuration data to or from the configuration image at run time.

To generate and manage configuration images and their content easily, they have the same format for all node types. A configuration image is a single loopback file containing a Second Extended Filesystem (Ext2), the standard file system used on Linux systems. This file system holds the current contents of the configuration image. On nodes with enough storage capacity the configuration image can be directly written to disk or flash, with enough space available it is even possible to install multiple configuration images, while on nodes with limited storage other techniques have to be used. On the Meraki platform for example, the contents of the configuration image are saved in a 1 MByte Journalling Flash File System version 2 (JFFS2) partition, as there is no more space left on the device. However, the only really important point is that a node can make the contents of its configuration image available by mounting them to */mnt/config*. With this approach all nodes see their configuration information at the same place not regarding the method the configuration image is stored on the node.

5.1.3 Standalone Images

Standalone images are fully functional as they are and run completely in RAM of the target platform. They are a special class of software images and require no additional configuration image to work. Static configuration data is embedded into standalone images, they contain the whole root file system plus the configuration data in their initramfs archive and are therefore fully self-contained. They are perfectly adjusted for testing node hardware without bricking it. It is not necessary to install a standalone image on the node hardware for a working WMN, these images are only used to test node hard- and software (mostly the kernel itself) in a simple and preserving way. Standalone images can even use a Network File System (NFS) root file system, when the boot loader supports providing a command line to the Linux kernel.

Some nodes require standalone images for installing the normal software images on the node's flash storage. For example the boot loader on Meraki platforms does not support writing files bigger than 5 MByte to its flash storage, so a standalone image has to be used for this purpose. A standalone image is configured to be easy accessible, it tries to get network setting using a DHCP client on the wired interface and acts as an adhoc wireless Network Address Translation (NAT) gateway on one wireless interface. A standalone image does not contain any keys, therefore it cannot be managed with the configuration framework. Like in all

other software images, every change in the root file system of a standalone image is not permanent over reboots.

5.2 Software Image Creation

Creating software images out of the cross compiled software is a relatively straightforward process for the end user. Like *build-tool* for the build system, a corresponding front-end for the image creation process, called *image-tool* is used. As seen before, the resulting software image has to be a bootable kernel image for the given platform and contains an initramfs archive with the root file system of the node. Creating such an image therefore consists of generating the initramfs archive first, and second, a recompilation of the platforms Linux kernel for embedding the previously generated new initramfs archive. The resulting Linux kernel image has then to be adjusted to a format recognised by the platforms boot loader. After this procedure the software image is ready for use and gets renamed to a standardised file name *<board>-image-<version>.bin.gz*. The details of the software image creation are also shown in Figure 5.1.

5.2.1 Initramfs Archive Creation

For generating the initramfs archive the cross compiled software as well as initramfs templates are used. These templates contain an initial directory structure for the root file system and files required to boot until a configuration image can be mounted. Initramfs templates are located in the *initramfs* directory. An initramfs archive is created by *image-tool* by the following steps:

1. Creation of temporary directory

   This directory is the root for the initramfs archive.

2. Installation of common initramfs templates

   Install the initial directory structure found in the *common* initramfs template to the temporary directory.

3. Installation of platform specific templates

   Copy files found in the platform specific initramfs template to the temporary directory. This action allows overwriting files from the *common* initramfs template with platform specific templates.

4. Installation of cross compiled software

   As seen before in Chapter 4, the build system installs all cross compiled software for a given target platform to the home directory of the corresponding build user in *target/bin*, *target/sbin* and *target/lib*. Copy the contents of these directories to the temporary directory and strip all debugging symbols out of

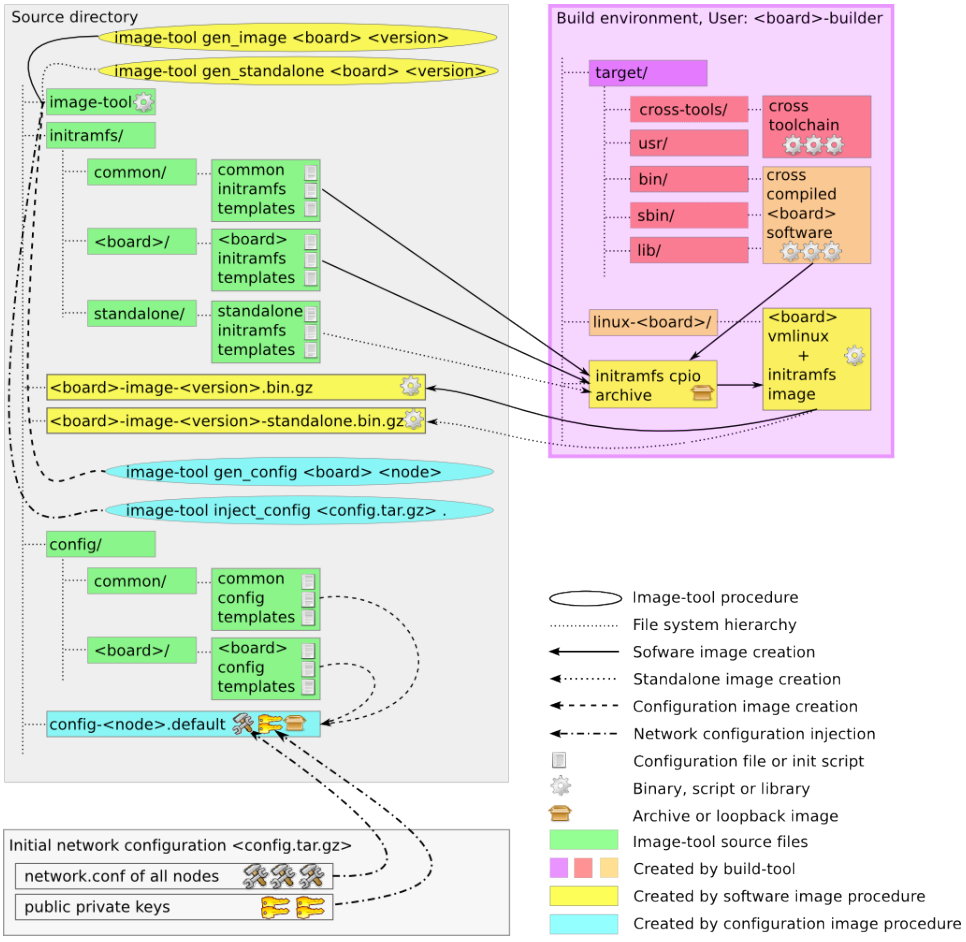


Figure 5.1: Details of the image creation process

the binaries, which makes them smaller and saves therefore valuable space in the software image.

5. Kernel module installation

   Install the kernel modules found in *target/lib/modules* to the temporary directory, compress them and adjust their dependency file. This saves a lot of space in the system RAM of the nodes at run time.

6. Initramfs archive generation

   Adjust ownership and permissions of all files in the temporary directory and generate a cpio archive out of it.

7. Installation of initramfs archive

   Move the new initramfs archive to the home directory of the platform build user, where it can be found by the kernel recompilation procedure.

5.2.2 Linux Kernel Recompilation

For embedding the newly created initramfs archive a recompilation of the platform specific Linux kernel image is needed. For each target platform this is done by a special package build script in the build system. This build script (e.g. *meraki-image.sh*, *alix-image.sh*) only recompiles the platform specific kernel image from the kernel sources in the build users home directory. Since a generic kernel image was already compiled by the build framework, the recompilation of the platform specific kernel image consists of relinking the already compiled objects and the new initramfs archive to a new platform specific kernel image.

5.2.3 Parameters for Image-Tool

For each software image to be created, *image-tool* needs to know the type of the image (normal or standalone) as well as the node platform the image is for. In addition a version string for the software image is required to distinguish different software images from each other. Figure 5.1 shows schematically the creation of software images out of the cross compiled software and initramfs templates. For completeness the output of creating software images with *image-tool* is shown in Listings 5.1 and 5.2 for a normal software and a standalone image, respectively.

5.3 Configuration Image Creation

Like software images, configuration images are created with the *image-tool* front-end. Unlike software images, configuration images contain no cross compiled software and therefore their creation is less complex. For creating configuration images only configuration templates are used. These templates are located in the *config* directory and structured like initramfs templates. A generic configuration template,

which is shared by all node types, is used to create an initial configuration image. The contents of this generic template, called *common*, can be individually overwritten for each node type by using a node type specific configuration template. The procedure is very similar to the one outlined for initramfs templates. Since configuration images are loopback files containing a Ext2 file system, *image-tool* has to create such an loopback file out of the configuration templates. In order to create a configuration image *image-tool* requires the node type for choosing the right template and the host name of the node, as each node has a dedicated configuration image. A sample output of *image-tool* for configuration image creation is provided in Listing 5.3.

A configuration image is only complete if an initial network configuration described in Section 6.4 has been included. Such a network configuration can be created manually or it is provided by the ADAM GUI in a compressed tar archive format, and contains all initial network configuration files and all public/private key pairs of all nodes. This initial network configuration is injected into config images with *image-tool*. Therefore *image-tool* needs the configuration image of every node participating in the configured network and the initial network configuration. Before injecting the initial network configuration into the configuration images, *image-tool* checks for the existence of the required configuration images, keys and *network.conf* files of the nodes. A sample output of *image-tool* for injecting an initial network configuration is provided in Listing 5.4. Figure 5.1 shows schematically the creation of a configuration image out of configuration templates and the injection of an initial network configuration.

5.4 Installation and System Booting

Once all necessary images have been created by *image-tool*, these images have to be installed on the node hardware. This means at least a software image for each node hardware platform, and a configuration image for each single node have to be available. Installing the images to the node hardware is called deployment and can be a rather complicated procedure depending on the platform used. Boot loaders have to be configured, second storage has to be partitioned, file systems have to be made and software as well as configuration images have to be installed to secondary storage. Due to the high hardware dependency of these tasks the deployment procedure differs heavily from one platform to another. Therefore automation is very limited. A README file is provided for each supported target platform, which contains an exact description of the particular hardware deployment process. For completeness two particular deployment processes are described in detail. The first method is designed for devices which can use GRand Unified Bootloader (GRUB) [26] and have block devices, which can be partitioned, like hard disks or compact flashes. GRUB is the default boot loader used by the most modern Linux distributions. A second method treats devices like the Meraki with

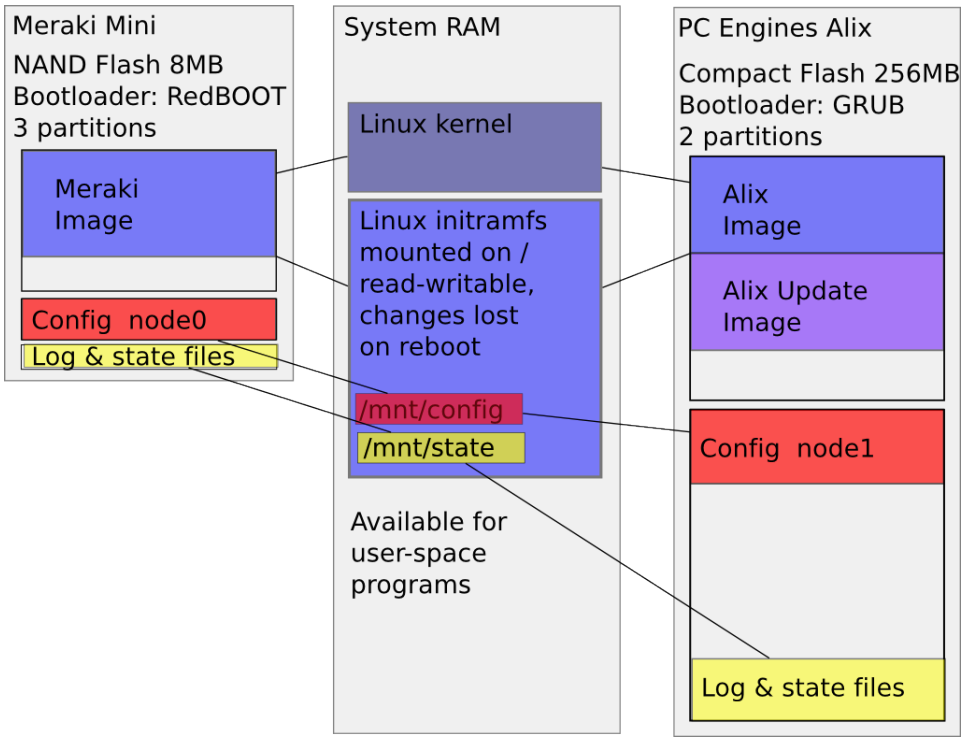


Figure 5.2: Run time layout of RAM and secondary storage of two platforms

custom boot loaders and internal NAND or NOR flash storage. Devices of the first category are preferable due to the possibility of using a safe update mechanism for software images with the help of GRUB, described in Subsection 6.6.1. Figure 5.2 shows a schematic run time layout of RAM and secondary storage of two different deployed target nodes. The layout of the RAM looks the same for both hardware platforms, which is intended.

5.4.1 GRUB with Normal Block Devices

If a platform can use GRUB and contains secondary storage known to the Linux kernel as a normal block device, like a hard disk, the deployment process is pretty straightforward. First of all, the block device has to be partitioned into two partitions. These partitions get formatted with an Ext2 file system. The first partition should be big enough to hold two software images if the safe update mechanism with GRUB described in Subsection 6.6.1 should be used. On both partitions GRUB configuration and stage files, which are used to read the file system have to be installed. On the second partition two directories *config* and *state* have to be made. After that, the software image is copied to the first partition in the / directory and the configuration image to the second partition in the *config* directory. In a last step GRUB has to be installed to the Master Boot Record (MBR) of the block device. Alix and WRAP nodes contain even an interchangeable compact flash card which allows performing the deployment procedure on another system than the node itself.

5.4.2 Custom Boot Loader with Flash Storage

The deployment process for systems with secondary storage realised in NAND or NOR flash storage devices can be very complicated. This kind of flash storage is known to the Linux kernel as Memory Technology Device (MTD) and treated as character based, not as block based device. The partitioning and formatting of such flash storage devices requires mostly special non standardised tools.

However, the Linux kernel allows treating such devices as block devices using a special block emulation layer. This enables writing data in blocks to NAND or NOR flashes with the dd tool, but introduces a big emulation overhead. There exist special file systems supported by the Linux kernel, e.g. JFFS2, which are tailored for MTD. These file systems provide to most efficient read and write operations on MTD. Therefore configuration image contents have to be saved in a JFFS2 file system on this device class.

Moreover, systems with NAND or NOR flash storage often use customised boot loaders. Some of them are able to partition the flash device, others are not, some require special Linux kernel image formats, others can boot normal gzip compressed vmlinux images. An example of a platform which uses internal NAND flash and a custom boot loader is the Meraki platform. The deployment process for this platform is done with the help of a standalone image containing special scripts

to install the configuration and software image on the device. The unsafe update mechanism for these devices is explained in Subsection 6.6.2.

5.4.3 Log and State Files

Each node uses special partition for the permanent storage of log and state files like random seeds or log files. Since log files are rotated on all nodes, the size of these log and state files can regarded to be constant. A special mount point on the nodes, */mnt/state* is used to access the secondary storage which contains the log and state files. The partition which is used for this purpose has to be created in the hardware deployment procedure, and can differ in its file system and its size between the different platforms.

5.4.4 Booting a Node

The boot process of a node involves several steps until the node is fully operational. Independent of the hardware type, if a node is powered up, the boot loader loads the software image into RAM. After having been executed by the boot loader, the Linux kernel takes over control, initialises hardware devices and mounts its embedded initramfs archive as root file system and tries to execute the */sbin/init* binary found on the root file system as process with Process IDentfier (PID) 1. This process starts various so called init scripts, located in the */etc/init.d* directory on the root file system, which take control of the further boot process. Figure 5.3 shows an overview of the init scripts involved, and their execution order.

The first init script executed by */sbin/init* is called *rc.sysinit* and responsible for mounting virtual file systems and populate the */dev* directory with device nodes. It executes *rc.config*, which loads the contents of the configuration image to the root file system. Therefore both scripts *rc.sysinit* and *rc.config* must be contained in the initramfs archive. All other init scripts are contained in the configuration image and can be used after its successful loading. The first of these init scripts executed is *rc.state* which makes available the permanent storage for log and state files on */mnt/state* as mentioned in the previous subsection. The second script *rc.services* is a dispatcher script for various other init scripts which start and stop services statically defined in the configuration file */etc/conf.d/rc.conf*. The default setup starts the syslog, network, dropbear, crond and httpd services, which are required for a functional configuration framework and remote logins. If the node boots the first time, keys for the dropbear Secure SHell (SSH) server and a Secure Sockets Layer (SSL) server certificate for httpd are automatically generated and stored in the configuration image by the init scripts. The special network init script is responsible for applying all network configuration manageable by the configuration framework. Therefore the configuration information contained in */etc/conf.d/network.conf* is used. The *rc.network* script reads the newest version of the */etc/conf.d/network.conf* available and executes the following actions:

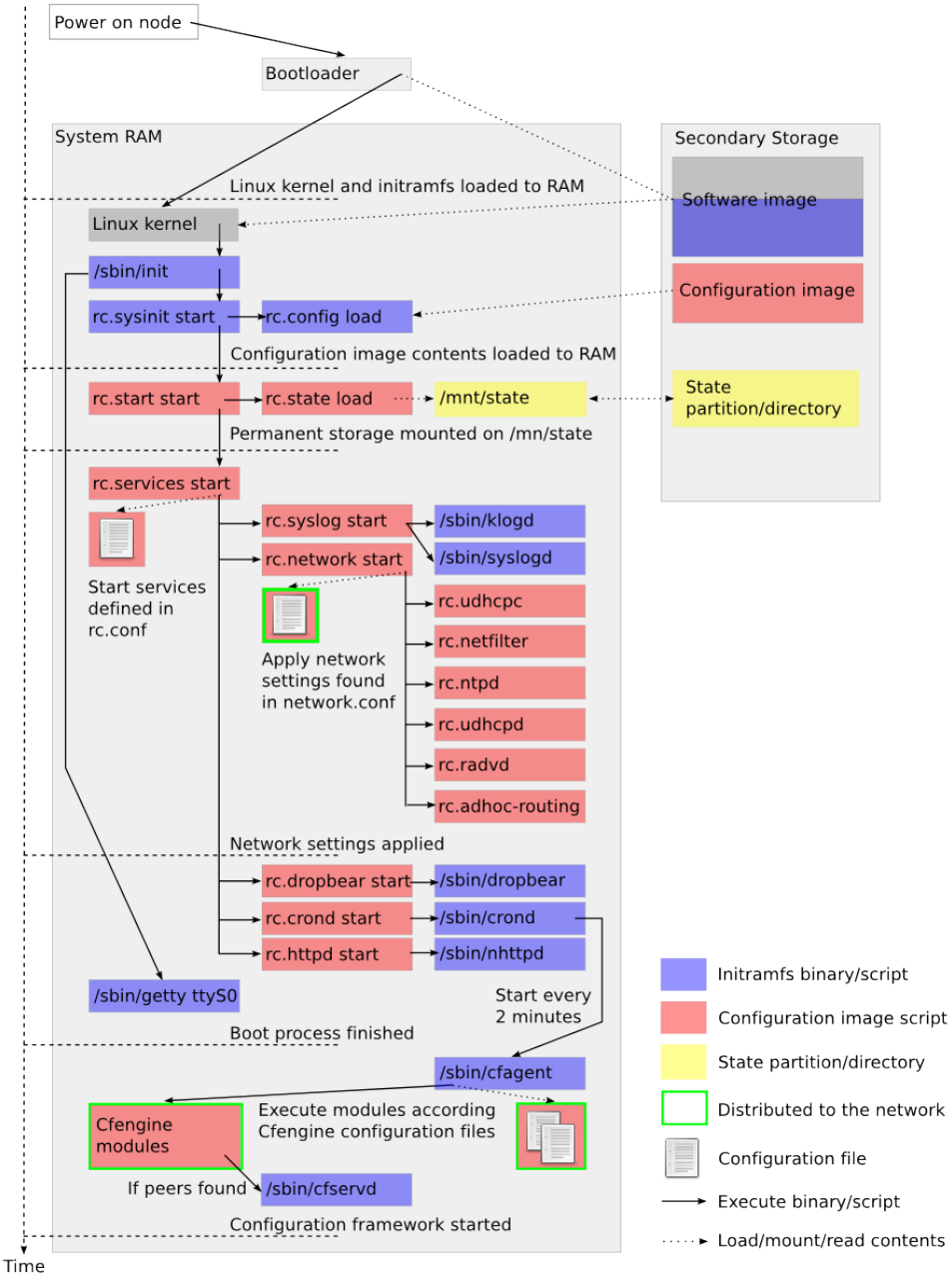


Figure 5.3: Details of the boot process

- Identify the network interfaces which have to be configured.
- Set configured netfilter firewall rules for IPv4 and IPv6.
- If some interfaces are configured by IPv4 DHCP, get the lease information and write it to */etc/conf.d/network.conf* as static information.
- Setup the special configuration IPv6 address described in Section 6.3.2 on the wireless interface known by its Media Access Control (MAC) address.
- Configure all static IPv4 and IPv6 addresses on the requested interfaces.
- Set static default routes for IPv4 and IPv6.
- Configure static host name resolving in */etc/hosts*.
- Configure static Domain Name System (DNS) servers in */etc/resolv.conf*.
- Generate the configuration file */etc/ntpd.conf* for NTP, and start the corresponding server or client if requested.
- Generate the configuration file */etc/udhcpd.conf* for the IPv4 DHCP server and start the service if requested.
- Generate the configuration file */etc/radvd.conf* of the IPv6 router advertisement daemon and start the service if requested.
- Configure adhoc routing services. This feature is only partly implemented at the moment

The *rc.network* script is also used by the network configuration module described in Subsection 6.5.1. After the script *rc.sysinit* has exited, the network is configured according */etc/conf.d/network.conf* and static services listed in */etc/conf.d/rc.conf* are started. The boot process has been completed and cfagent is started periodically all two minutes by crond. After its first execution a node is fully functional and self-contained and can be managed by the ADAM configuration framework described in Chapter 6.

5.5 Sample Output of Image-Tool

This section provides some sample output of *image-tool* during the creation of software and configuration images and the application of the initial network configuration to the configuration images.

```
root@monk:~/image-builder# ./image-tool gen_image meraki 205
20441 blocks

This will install the following packages in order from left to right:

meraki-image

Packages will be built in:     /home/meraki-builder
Package sources are in:        /home/meraki-builder/sources
Extra configuration is in:     /home/meraki-builder/buildconfig
Packages will be installed to: /home/meraki-builder/target
Packages get compiled by:      /home/meraki-builder/buildscripts/packages
    /<package>.sh
Buildlog is in:                /home/meraki-builder/buildlogs/<package>.
    buildlog
Cross-Compiler used:           mips-linux-uclibc-gcc
Target system triplet is:      mips-linux-uclibc

Are these values reasonable ? Begin installation [y/n]
y

Wed Apr  1 16:39:43 CEST 2009: Installing package meraki-image...

Wed Apr  1 16:39:53 CEST 2009: Installation of packages meraki-image
    completed.

root@monk:~/image-builder#
```

Listing 5.1: Output of the software image creation process

```
root@monk:~/image-builder# ./image-tool gen_standalone meraki 205
20743 blocks

This will install the following packages in order from left to right:

meraki-image

Packages will be built in:     /home/meraki-builder
Package sources are in:        /home/meraki-builder/sources
Extra configuration is in:     /home/meraki-builder/buildconfig
Packages will be installed to: /home/meraki-builder/target
Packages get compiled by:      /home/meraki-builder/buildscripts/packages
    /<package>.sh
Buildlog is in:                /home/meraki-builder/buildlogs/<package>.
    buildlog
Cross-Compiler used:           mips-linux-uclibc-gcc
Target system triplet is:      mips-linux-uclibc

Are these values reasonable ? Begin installation [y/n]
y

Wed Apr  1 16:38:35 CEST 2009: Installing package meraki-image...

Wed Apr  1 16:38:49 CEST 2009: Installation of packages meraki-image
    completed.

root@monk:~/image-builder#
```

Listing 5.2: Output of the standalone image creation process

```
root@monk:~/image-builder# ./image-tool gen_config  meraki meraki0
root@monk:~/image-builder# ./image-tool gen_config  meraki meraki1
root@monk:~/image-builder# ./image-tool gen_config  meraki meraki2
root@monk:~/image-builder# ./image-tool gen_config  alix alix0
root@monk:~/image-builder#
```

Listing 5.3: Output of the creation of several configuration images

```
root@monk:~/image-builder# ./image-tool inject_config testcfg.tar.gz .
root@monk:~/image-builder#
```

Listing 5.4: Output of the injection of an initial network configuration

Chapter 6

ADAM Configuration Framework

This chapter focuses on the ADAM configuration framework which is used to configure the particular nodes in the network after deployment. After deploying node hardware, each node has a software image matching its architecture and a configuration image injected with an initial network configuration installed. Henceforward each node can be physically installed to its dedicated location, and is then managed remotely by the ADAM configuration framework without needing further physical access to it. The newly implemented ADAM configuration framework is a completely redesigned solution of some basic ideas found in SRM [16], it is more flexible due its modularity, has full support for IPv6, shows faster propagation times for network configuration changes and features a more stable peer detection mechanism.

6.1 Requirements

The redesigned ADAM configuration framework should therefore meet the following requirements:

- The configuration framework aims to be fully distributed and decentralised. It should be possible to configure any node in the network from any other arbitrary selected node.
- Nodes should never loose connectivity to each other. Only physical circumstances, not misconfiguration are responsible for loosing connectivity to a node.
- The web based ADAM GUI should provide only an extension to offer a more user-friendly way to the management framework than editing text files on a normal node which has no GUI installed. The configuration framework does not make differences between normal and management nodes, in terms of functionality, all nodes in the network behave the same.
- All communication has to be encrypted and only be allowed between authenticated nodes in the network.

- It is very important for the configuration framework not to rely on any external dependency for a proper functionality. Such dependencies include time sources like NTP servers or hardware clocks, name services like Domain Name System (DNS) servers and automatic address configuration systems like DHCP. Nevertheless it should be possible to use such external dependencies on demand without disturbing the ADAM configuration framework.
- The framework has to be modular and extensible in terms of the manageable parameters on the nodes. It should be easily possible to redefine or add configuration parameters and the corresponding actions.
- The configuration framework has to be portable since it has to run on different node hardware platforms.
- All shell scripts used in the configuration framework have to be written compatible for the Busybox based ash shell used by the nodes. Since ash syntax is Bash compatible but not vice versa, all shell scripts in the configuration framework are written in ash syntax to be compatible with both shells.
- The ADAM configuration framework has to fully support IPv6 in addition to IPv4. This requires all networking tools, clients and servers being both IPv6 and IPv4 capable.

6.2 Overview

Like the SRM configuration system the ADAM configuration framework architecture is based on Cfengine [15]. The ADAM configuration framework is implemented from scratch using Busybox shell compatible scripts installed on the nodes.

All configuration framework communication between the nodes is handled by the Cfengine server cfservd and the corresponding agent cfagent. Communication of these two components is encrypted with a public/private key mechanism. ADAM is configured to allow only encrypted traffic between cfagent and cfservd. In addition, a node needs all public keys of all other nodes before communication with their cfservd is possible. Therefore ADAM allows communication only to already authenticated nodes, whose public key is already available.

ADAM uses a dedicated IPv6 network according Request For Comment (RFC) 4193 [27] for all Cfengine connections. Addresses in this network are calculated from a given prefix and the Media Access Control (MAC) address of the node's network interface. In the ADAM architecture, each node knows this common prefix and one MAC addresses of all other nodes. Therefore the address of each other node in the dedicated IPv6 network can be calculated and used for peer detection.

On each node cfagent fetches periodically the newest configuration files from all detected peers. Once the newest configuration files are available, cfagent processes their contents and configures the node accordingly. The

ADAM configuration framework allows configuring IPv4 and IPv6 network settings and services of all nodes on any arbitrary selected node in the network and is therefore fully distributed. Additionally the ADAM configuration framework allows updating software images, extending existing ADAM WMNs with new node hardware and executing user defined commands on selected nodes in the network.

6.3 Core Architecture

The provided implementation of the ADAM configuration framework fulfils all the requirements shown in Section 6.1. Being fully distributed and autonomous, every participating node offers exactly the same minimal functionality. The core components used on the nodes are cfservd, the server part and cfagent the agent which handles all current configuration actions and is executed periodically by the cron daemon. Every time cfagent is executed by cron on a particular node, it connects to the cfservd of every reachable peer and fetches new configuration information from them. This new configuration information is then processed by the different ADAM configuration modules invoked by cfagent, described in detail in Section 6.5. In addition, an SSL and IPv6 capable web server and client are required for the special system time synchronisation procedure required to guarantee a proper functionality of the ADAM configuration framework.

For secure communication cfagent and cfservd use an asymmetric public/private key mechanism. In the actual implementation cfagent and cfservd are configured to allow connections only if the corresponding keys are already known to the node. This setup makes it impossible to join a network without having the required keys and therefore prevents from man-in-the-middle attacks.

6.3.1 Interaction of Cfagent and Cfservd

The principle of interaction between cfagent and cfservd on the different nodes is very simple. On each node cfagent is executed periodically all two minutes by the cron daemon, connects to cfservd of all currently reachable nodes in its neighbourhood. The procedure is shown in Figure 6.1 for one dedicated node in the network. Every time cfagent is executed it performs the following actions:

1. Wait a random time period between 0 and 60 seconds. This prohibits the different nodes from connecting all at the same time and flattens out network traffic peaks.

2. Detect reachable peers. This is described in more detail in Subsection 6.3.3.

3. Synchronise the local system clock with the newest value found on all reachable peers. The detailed procedure and why it is crucial for a working solution is outlined in Subsection 6.3.4.

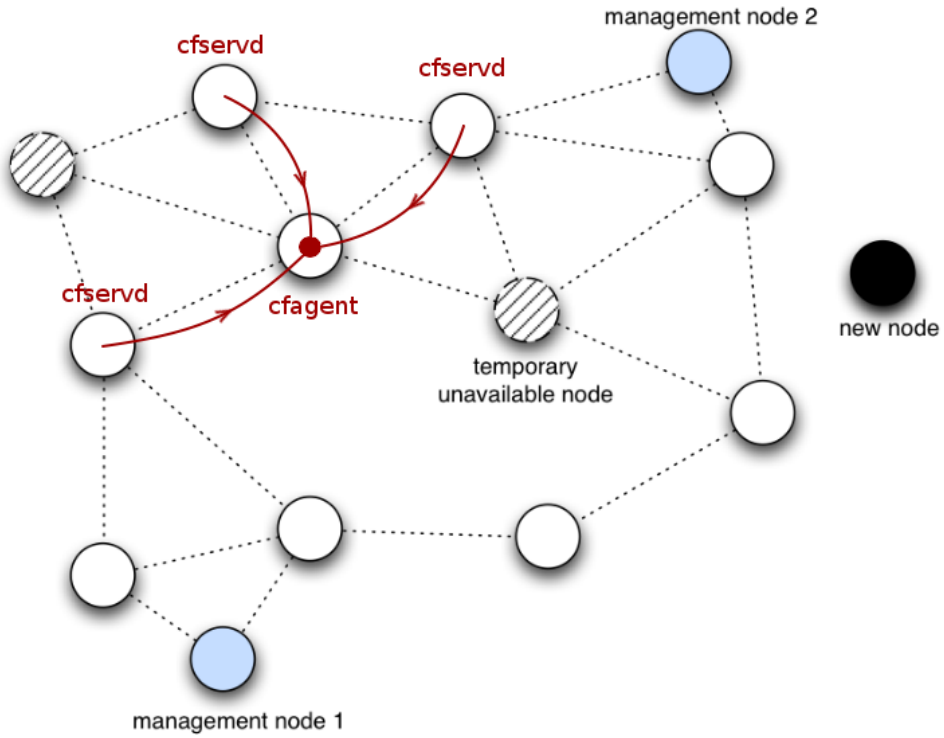


Figure 6.1: Interaction of cfagent and cfservd

4. If at least one peer could have been reached, start the local cfservd to allow future connections from the remote peer's cfagent. If no peer is found stop an eventually running local cfservd.

5. Connect to all reachable peers cfservd and check for a newer Cfengine configuration. If such a configuration is found, apply it.

6. Once again connect to all reachable peers cfservd and get the newest configuration information found among them.

7. Run the various local configuration modules according to the newest configuration information available. The configuration modules are described in Section 6.5.

8. Clean up old and useless files and temporary directories, check and correct local file permissions and verify the existence of vital processes (e.g. crond, httpd).

6.3.2 Dedicated IPv6 Cfengine Network

The dedicated IPv6 network for Cfengine consists of an IPv6 prefix, a so called Locally Assigned Global ID according to Internet Engineering Task Force (IETF) RFC 4193 [27]. This prefix is generated once for each network according to the algorithm described in the RFC and it is the same for all nodes participating in the network. To calculate the IPv6 address used for the dedicated network, the MAC address of one wireless interface per node and the previously generated IPv6 prefix is needed. The resulting address is a Unique Local IPv6 Unicast Address according to RFC 4193. If every node knows this previously generated IPv6 prefix for the configuration network and the MAC address of a wireless interface on each node, it can calculate the dedicated IPv6 address of each other node on the fly, which makes peer detection simpler and much more stable, than depending on a manual network configuration.

6.3.3 Reachable Peer Detection

ADAM introduces a peer detection which is fully independent of the current configurable network settings of a node. Every node participating in the configuration framework has configured an IPv6 address according the technique described in the previous subsection. The configuration framework guarantees that each node has at least one interface configured for this special IPv6 network at any time. Since all nodes are aware of these special IPv6 addresses of each other (they can be calculated from the known prefix and the MAC address of a node), the dedicated IPv6 network is used for cfagent and cfservd connections. Every node has at least one address configured for the dedicated IPv6 network any time, therefore unreachable nodes due to wrong network configurations are impossible. Only physical cir-

cumstances (e.g. insufficient signal strength) can be responsible for not reaching a particular node.

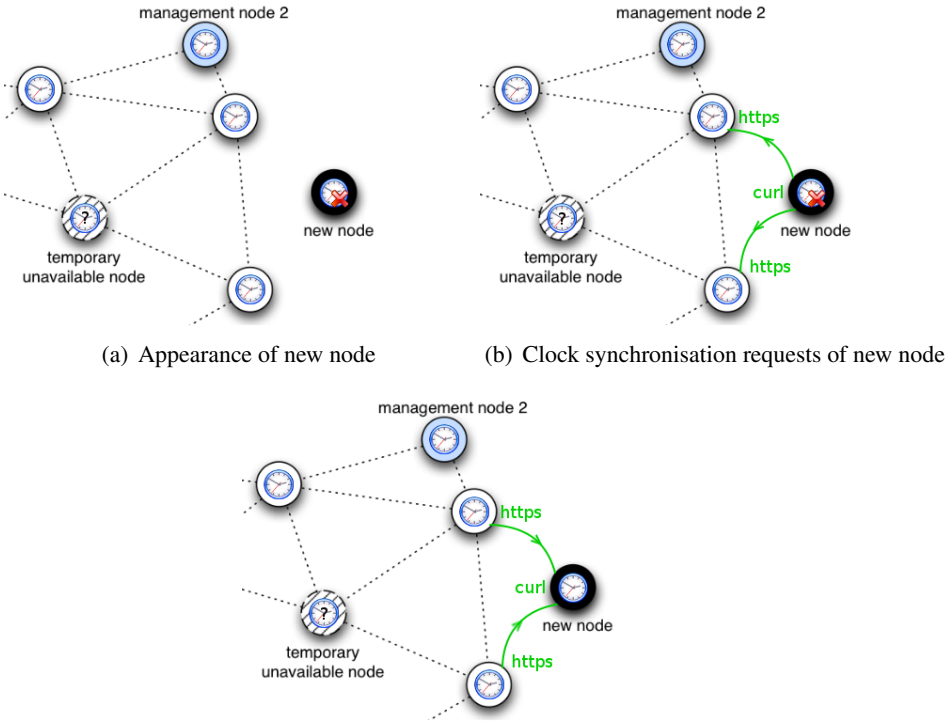
Using the dedicated IPv6 network addresses reachable peers can be detected very easily. All possible peers (all nodes in the network) are known through their special IPv6 addresses in the dedicated IPv6 network. Detecting the currently reachable peers from the group of all possible peers is simply done by tracerouting their address in the dedicated IPv6 network. Normally this peer detection with traceroute works over one hop, but can be configured to detect peers more than one hop away, if routing is set up accordingly. This outlines a major advantage of the new ADAM solution by which no routing has to be set for a working configuration framework. This actually allows even the configuration of routing mechanisms through the configuration framework, without influencing the framework itself.

Since it is easily possible to assign multiple addresses to a single network interface in Linux, this new solution has only the same limitations in network configuration possibilities as the usual physical constraints given for wireless communication. The communicating parties have to share the same Extended Service Set IDentifier (ESSID), the same channel and the transmission power has to be sufficiently high. For issues with such wireless settings and their influence on configuration framework connectivity see Subsection 6.3.6. The new solution has therefore no more need for a time consuming and complicated detection mechanism for lost nodes like used in SRM. In addition, the new solution allows automatic address configuration for all interfaces on all nodes in the network and therefore no static addresses have to be known at deployment time.

6.3.4 System Clock Issues

Both cfagent and cfservd are very sensitive on system clock differences between two connecting nodes. Very sensitive means, differences bigger than one minute influence the proper functionality of the used Cfengine configuration, differences much bigger than a minute can even prevent cfservd from accepting incoming connections at all. This could be for example the case when a new deployed node without hardware clock boots (its system clock is probably set back in January 1970), and wants to connect to nodes in the network which have a correct system clock. Due to the requirements of complete autonomy and a distributed design, synchronisation with external NTP servers is no possible solution, as no successful clock synchronisation may be guaranteed in any case. Synchronisation with battery driven hardware clocks is no adequate solution as well, since some nodes may actually not even have such a hardware device. Therefore a hardware and Cfengine independent solution for system clock synchronisation between the nodes is required.

The synchronisation solution is implemented with a Common Gateway Interface (CGI) script running on the SSL and IPv6 capable web server, and a corresponding client for connecting to the CGI script. Figure 6.2 (a) shows a newly



(a) Appearance of new node

(b) Clock synchronisation requests of new node

(c) Clock synchronisation replies of two peers

Figure 6.2: Synchronising system clocks between nodes

appearing node, which has no valid system clock. Each node connects with the curl client to the web server of all its peers, shown for the new node without a valid clock in Figure 6.2 (b). The CGI script on each peer returns the actual system clock as UNIX time stamp and the sha1 hash of this time stamp combined with the public key of the node the CGI script is executed on. This is shown for the peers of the new node without valid clock in Figure 6.2 (c). According to the received hash value a client node decides if the reply is from an authorised node or not. This mechanism prevents from man-in-the-middle attacks. If the received hash value belongs to an authorised node, the local system clock is set to the received time stamp only if it is bigger than the one calculated from the local system clock. This policy ensures that the system clocks are monotonic increasing functions in time. A node synchronises its local system clock therefore always to the newest system clock found among all its peers.

Since system clocks can drift to the future, a configurable parameter *DRIFTTIME*, which is set in the *network.conf* file of each node defines the number of seconds a clock can maximally drift to the future between two synchronisation requests. The local system clock is set only to a maximum value of the received remote time stamp minus the value of *DRIFTTIME*. Since time synchronisation is done with all remote peers every time cfagent is executed (all two minutes), a value of one second is generally enough for *DRIFTTIME*. Without this mechanism, one system clock drifting to the future would lead to a drifting system clock on all nodes in the network. But the mechanism implies an additional difference of *DRIFTTIME* seconds between the system clocks of two different nodes. With values less than five seconds for *DRIFTTIME*, there is no influence on Cfengine connections, and no system clock should drift more than five seconds in two minutes. The chosen implementation takes also into account, that a local system clock has to be synchronised before connecting to any other node with cfagent and before accepting any connections to cfservd from other nodes.

6.3.5 Configuration Distribution

Cfagent and cfservd provide all the functionality for distributing the whole configuration. Even the configuration of the cfagent and cfservd components themselves can be altered on the fly for the whole network. Configuration distribution with cfagent for either Cfengine components or for the configuration framework is achieved by getting the newest files from cfservd of all actual remote peers. This mechanism behaves like a directory or file shared in the network, where every node has a local copy of it, which is overwritten always with the newest version found among all remote peers. Therefore removing a file in such a directory causes cfagent to fetch it again from another peer. When such a file has really to be removed, it is enough to replace it with a zero size one and cfagent will remove it when it is older than a day. The configuration distribution and the files and directories involved are shown in Figure 6.3. Here is a list of current directories shared on each node and

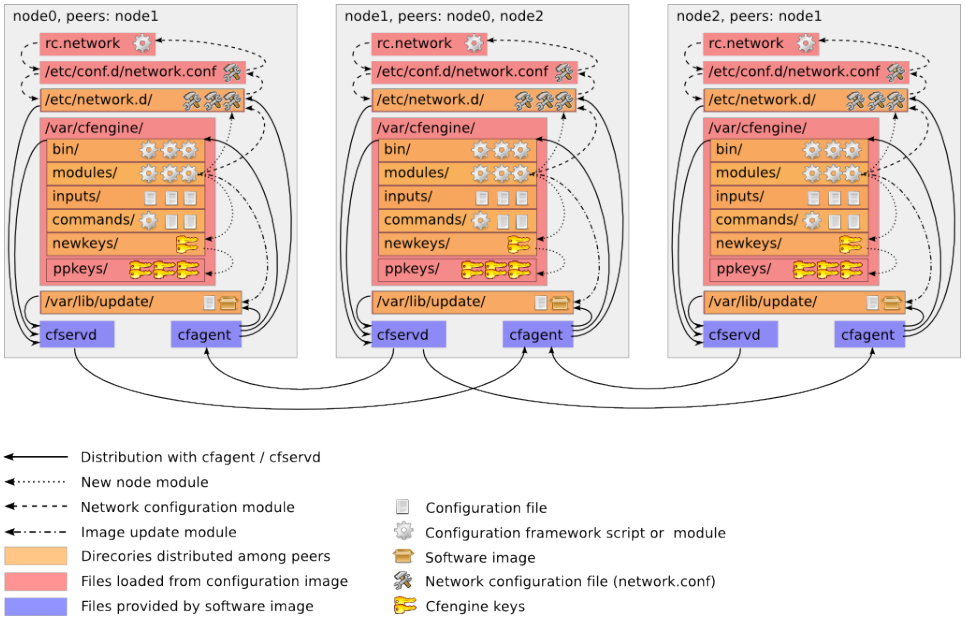


Figure 6.3: Configuration distribution with cfagent and cfservd.

the purpose for their sharing.

- The directories */var/cfengine/*{*inputs,modules,bin*} are used for internal configuration of cfagent and cfservd and contain the configuration modules themselves.
- */etc/network.d* contains all network configuration used by the network configuration module described in Subsection 6.5.1.
- */var/cfengine/newkeys* contains public keys of new nodes used by the new node module described in Subsection 6.5.2.
- */var/cfengine/commands* contains commands and their data used by the command module described in Subsection 6.5.4.
- */var/lib/update* contains new software images and update information which is used by the image update module described in Subsection 6.5.3.

6.3.6 Detection of Misconfigured Nodes

There are two possible scenarios where a node cannot reach its peers anymore, even when its network interface is configured properly for participating in the dedicated IPv6 Cfengine network introduced in Subsection 6.3.2. First, the administrator may manually set a too low transmission power manually, which could cause the

loss of connectivity to all peers. Second, the administrator can configure another wireless protocol (802.11a, b or g) than the one used by all other peers.

These settings cannot be automatically adjusted by the network configuration scripts. Therefore a simple detection mechanism for such misconfiguration is used. When a node reaches no peers within a timeout defined in */etc/conf.d/network.conf*, it regards itself as misconfigured and resets the transmission power and wireless protocol to sane values. The default value for the timeout is 24 hours, transmission power is reset to an automatic maximum and the wireless protocol set to 802.11g, which all nodes support. Except for these two scenarios, it should not be possible to loose connectivity to the configuration framework in any way, since one interface is always configured for participating in the dedicated IPv6 Cfengine network.

6.4 Initial Network Configuration

As seen in Subsection 6.3.5 each node participating in the ADAM configuration framework shares the */etc/network.d* directory with all other nodes. This directory includes a configuration file for each node containing all the parameters relevant for the ADAM configuration framework. The collection of all these node configuration files, the public/private key-pair of all nodes and a root password for the nodes is called a network configuration. An initial network configuration is provided in form of a single gzip compressed tar archive, which is used by *image-tool* to inject configuration images with it, as explained in Chapter 5. An initial configuration archive can be created on each node with a text editor and the Cfengine cfkey tool, or it is created by the ADAM GUI on a management node. The injection mechanism is used to install an initial network configuration into the configuration images before deploying the node hardware. This procedure ensures that all nodes have only their own private key, but all public keys and configuration files of all other participating nodes installed at deployment time, which are needed for a working configuration framework. After this point all node configuration actions like changing network and services parameters, image updates, custom commands and adding new nodes are made with help of the configuration modules.

6.5 Configuration Modules

The configuration framework consists of different modules which are executed by cfagent according to the newest configuration information available in the shared directories on the nodes. This modular architecture allows writing new modules for new configuration tasks rapidly and integrate them in the configuration framework easily. In the current implementation four modules are used for configuration tasks on the nodes, which are explained the following.

6.5.1 Network Configuration Module

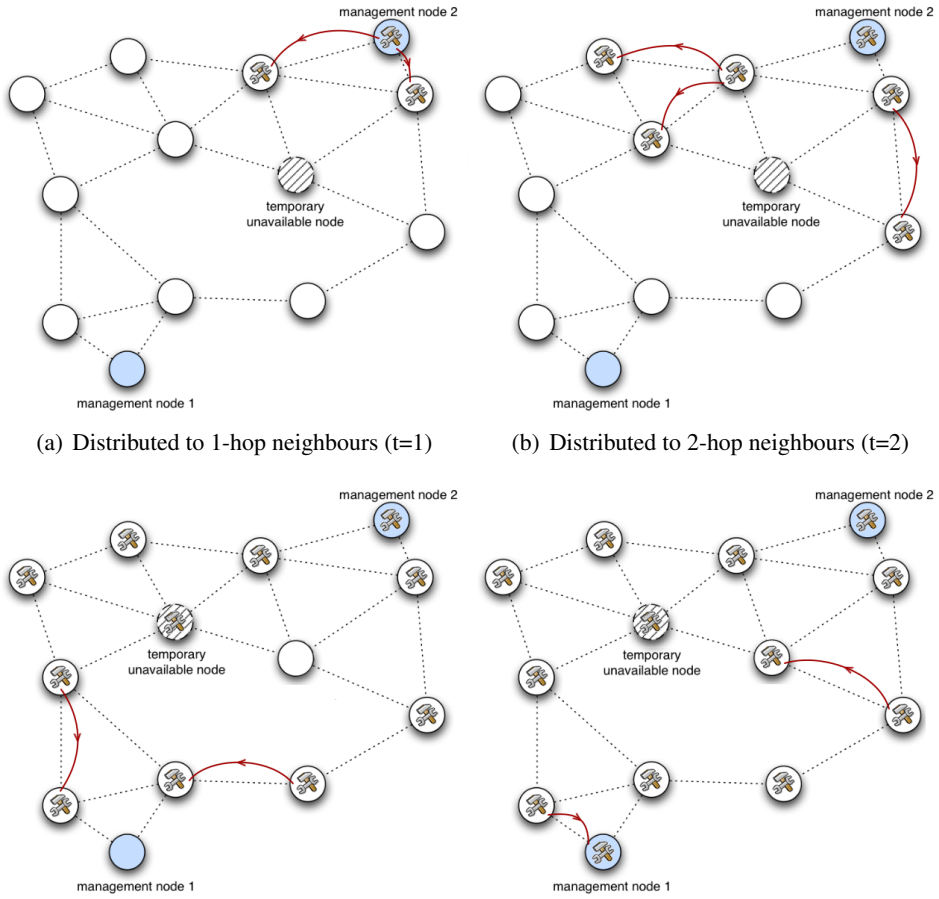
The network configuration modules is the most important module used in the framework. It is responsible for configuring and adjusting the settings of each node according the parameters defined in the node's configuration file found in the shared directory */etc/network.d*. These configuration files follow a simple naming convention and are called */etc/network.d/<node>.conf*, where <node> is the host name of the node. In addition, each node has a non-shared local copy of its configuration file in */etc/conf.d/network.conf* which represents the current configuration, which the node has applied at the moment.

The network configuration module detects differences between the actual applied parameters from */etc/conf.d/network.conf* and parameters defined in a new node configuration file found in */etc/network.d/<node>.conf*. Detected changes can be applied immediately by the module, since the connectivity with the configuration framework is not influenced. The configuration file of every node can be changed in the */etc/network.d* directory on any arbitrary selected node. This file is then newer than the corresponding one found on any other remote node, and the desired changes are propagated through the network automatically with help of cfagent and cfservd.

This works also for dynamic settings like for example DHCP acquired IPv4 addresses. If a node receives for example a new parameter in */etc/network.d/<node>.conf* to configure the IPv4 address on one interface with help of an external DHCP server, it can even react on the answer of the DHCP server. If a lease was obtained, the lease values are written back to */etc/network.d/<node>.conf* and shared in this way with other nodes.

The propagation of these network configurations is shown in Figure 6.4, where a new network configuration is defined at a management node in */etc/network.d*. Figure 6.4 (a) shows the situation after 2 minutes, where each cfagent of the 1-hop neighbours has connected to cfservd of the management node. After a further step all 2-hop neighbours of the management node have fetched the new network configurations as shown in Figure 6.4 (b). The network configurations are propagating from node to node with help of cfagent and cfservd, and are immediately applied by the network configuration module on each node. The propagation after four steps (eight minutes) is depicted in Figure 6.4 (c). The distribution has been completed after n steps in a network with depth n, shown in Figure 6.4 (d) for the network with $n = 5$.

The node configuration file found in */etc/network.d/<node>.conf* and */etc/conf.d/network.conf* respectively contains all configurable parameters for a node, like static and dynamic network settings and service configuration. The format of the configuration file is plain text and consists of comment lines beginning with # and assignments in the form *VARIABLENAME="value"*. For completeness, a sample configuration file for an Alix node is shown in Section 6.8 at the end of this chapter. At the moment the network configuration module allows configuring the



(a) Distributed to 1-hop neighbours (t=1)

(b) Distributed to 2-hop neighbours (t=2)

(c) Distributed to 4-hop neighbours while node availability changes (t=4)

(d) Distribution complete (t=5)

Figure 6.4: Network configuration distribution

following parameters for a node:

- Static IPv4 and IPv6 address configuration.
- Static name resolving for statically assigned IPv4 and IPv6 addresses.
- Definition of static IPv4 and IPv6 default routes.
- Definition of static external Domain Name System (DNS) servers for name resolution.
- Definition of static external NTP servers for time synchronisation.
- Dynamic IPv4 address configuration with DHCP. Even DNS servers, default IPv4 routes and NTP servers obtained by DHCP are supported.
- Dynamic IPv6 address configuration with router advertisement daemon.
- Configuration for a DHCP server running on the node.
- Configuration for an NTP server running on the node.
- Configuration for adhoc routing daemons. At the moment only a statically configurable olsrd variant is supported.
- Configuration for IPv4 and IPv6 netfilter firewall including IPv4/IPv6 forwarding and NAT for IPv4.

6.5.2 New Node Module

As described in Section 6.3 the used configuration of Cfengine only allows communication between nodes whose public key is already known to the communication partner. The integration of newly deployed nodes, whose public key and node configuration file are not yet known in the network is done with the new node module. If new node hardware is purchased and should be integrated into an existing network, first of all a software and configuration image has to be created for the the new nodes. In addition, the initial network configuration has to be extended with the new key pair and the configuration file for the new node, before injecting it to the configuration image for the new node. After injection the software and configuration images have to be installed to the new node hardware. Now the new node knows all other nodes by the keys and configuration files in its configuration image, but none of the nodes in the network knows the new node, as its public key and configuration file are not available in the network. Distributing these new public keys and node configuration files is the job of the new node module.

Distributing new configuration files is pretty straightforward, the new file can simply be copied to the */etc/network.d* directory, where it gets distributed to the network. The only thing the new node module has to do is to check if every node

configuration file in */etc/network.d* is available in the configuration image of the node too, in case the node reboots or reloads its configuration image.

The same thing happens with new public keys and the directory */var/cfengine/newkeys*. For security reasons, not the real key directory */var/cfengine/ppkeys* is shared over the network. Existing key files in the real key directory are never overwritten with a new key from */var/cfengine/newkeys*, only non-existing keys are installed. Checking for existence of all public keys in the configuration image has to be done in the same way as for node configuration files.

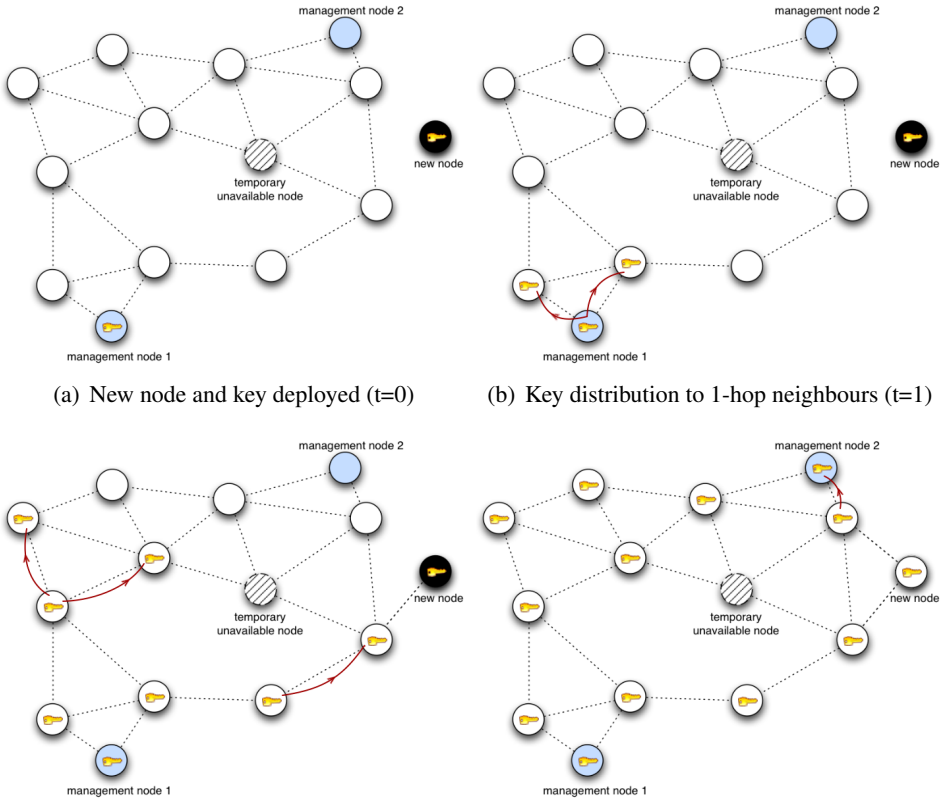
The procedure of adding a new node whose key and configuration file are not yet known in the network is shown in Figure 6.5. Figure 6.5 (a) shows the moment when the new public key and node configuration file are placed on the management node for distribution. After one cycle (2 minutes) all peers of the management node have fetched and saved the new files. This is shown in Figure 6.5 (b). The propagation of the new files is done with cfagent and cfservd. When the new files reach a direct neighbour of the newly deployed node, this direct neighbour will be able to connect to the new node in the next cycle, as shown in Figure 6.5 (c). After all nodes have fetched the new key and network configuration files, the new node can be regarded as fully integrated, since all nodes can possibly connect to the new node. This situation is depicted in Figure 6.5 (d).

6.5.3 Image Update Module

The image update module is used to install new available software images on the nodes. New software images are distributed to the network in the shared */var/lib/update* directory together with a file called *update*. This file contains the detailed information for the specific update action, like node type, update version and a sha1 hash of the image file. The module checks if the new software image announced in */var/lib/update/update* matches the node type it is running on. If the correct node type and version are detected, the image update module calls one of the node type specific update procedures described in Section 6.6.

New software images for different node types should be distributed in a serial way, avoiding that one node has to hold more than one software image at once in RAM, which is not possible on small nodes like the Meraki. Therefore the file size of existing images in the */var/lib/update* directories has to be set to zero by the administrator or the GUI, before copying a new image to this directory. For this reason only one software image for one node type can be distributed and installed with one update action at the same time. When all nodes of a specific type were updated, an new update action for another type can be issued.

Configuration images are node specific, each node has its own configuration image. Moreover, configuration images are stored in a different way on each node platform due to the different secondary storage used on it. Configuration images of all nodes are very similar, the differences between two configuration images of two nodes are little compared to their size (4 MByte). For these three reasons



(a) New node and key deployed (t=0)

(b) Key distribution to 1-hop neighbours (t=1)

(c) Key reaches neighbour (t=3)

(d) New node fully integrated (t=5)

Figure 6.5: Integration of a new node to an existing network

configuration images are not distributed in whole like software images.

The configuration images installed on the nodes have therefore to be altered in another way than distributing whole new configuration images for every node in the network. Adding, removing or changing files in the configuration images on the nodes is done with the command module described in the next subsection. Changing configuration image contents using the command module reduces network overhead compared to distributing whole configuration images. If a new file has to be added to all configuration images, the command module only distributes the new file and the shell command, which saves the file to the configuration image. The generated traffic is negligible compared to distribute a 4 MByte configuration image for every participating node.

6.5.4 Command Module

With the three modules already described in the last subsections, network configurations, software image updates and integration of newly deployed nodes is done. These three modules fulfil a clearly defined role and are difficult to extend. For being as flexible as possible the ADAM configuration framework provides a general purpose configuration module, the command module. The command module is used to execute arbitrary user-defined commands on a predefined set of nodes. Such a command can be specified with a command definition file, which contains the command string and a selection of nodes by their type or host name. These command definition files have the file extension *.cmd* and are distributed over the network in the */var/cfengine/commands* directory. Moreover it is possible to supply user defined data to a command definition file and distribute it over the network in the */var/cfengine/commands/data* directory. The command string can be either a binary already available on the nodes, or even a program or script contained in the user defined data.

When the command module is executed on a particular node by cfagent, it inspects each found command definition file, decides if the command has to be executed on the node, and executes the commands defined, if it was not executed already. If a command is executed on a node by the command module, a file named *node-command.reply* in */var/cfengine/commands/reply* is created, containing the execution time, the exit status and possible output of the command.

The command module is very powerful and can be used for a wide variety of administration actions, like for example distributing new configuration files or init scripts to the nodes or their configuration images, as described in the previous subsection. An example of a command definition file is shown in Listing 6.1. This command definition file uses a script as command string, which is contained in the user defined data. The script copies a sample init script and sample configuration file, which are also contained in the user defined data, to the nodes and their configuration images. The script is shown in Listing 6.2. A corresponding node reply file is shown in Listing 6.3.

```
# this defines a command to execute once on defined nodes
COMMAND="${DATADIR}/command.sh"

# the data directory related to this command is internally
# defined and generally not needed to redefine.
# DATADIR=/var/cfengine/commands/data/<cmd>.data

# run on these nodetypes
RUN_ON_TYPES="alix"

# run on these specific nodes
RUN_ON_NODES="meraki0"
```

Listing 6.1: Example of a command definition file (*test.cmd*)

```
#!/bin/sh

# This is an example for copying an initscript as well as a configuration
# file to the nodes root file system and to the configuration image.

# copy to root fs
echo "Copying files to root filesystem..."
install -m 755 ${DATADIR}/rc.initscript /etc/init.d/
install -m 644 ${DATADIR}/sample.conf /etc/
echo "Proof of existence after copy:"
ls -la /etc/sample.conf
ls -la /etc/init.d/rc.initscript

# copy to configuration image
mount /mnt/config
echo "Copying files to configuration image..."
install -m 755 ${DATADIR}/rc.initscript /mnt/config/etc/init.d/
install -m 644 ${DATADIR}/sample.conf /mnt/config/etc/
echo "Proof of existence after copy:"
ls -la /mnt/config/etc/sample.conf
ls -la /mnt/config/etc/init.d/rc.initscript
umount /mnt/config
```

Listing 6.2: Example of a script executed on defined nodes (*command.sh*)

```
Command: /var/cfengine/commands/data/test.data/command.sh
Node: meraki0
Date: Sun Apr 26 23:43:27 CEST 2009
Output:
Copying files to root filesystem...
Proof of existence after copy:
-rw-r--r-- 1 root root 42 Apr 26 23:43 /etc/sample.conf
-rwxr-xr-x 1 root root 37 Apr 26 23:43 /etc/init.d/rc.initscript
Copying files to configuration image...
Proof of existence after copy:
-rw-r--r-- 1 root root 42 Apr 26 23:43 /mnt/config/etc/sample.conf
-rwxr-xr-x 1 root root 37 Apr 26 23:43 /mnt/config/etc/init.d/rc.
    initscript

Exitstatus: 0
```

Listing 6.3: Example of a node reply file (*meraki0-test.reply*)

6.6 Updating Software Images

Updating software images on all nodes is controlled by the image update module mentioned in Subsection 6.5.3. However, the real installation procedure triggered by the module is different on each node platform. It has to take into account the platforms boot loader and secondary storage used and the technical way the software image has to be installed to it. Mainly there are two different procedures implemented on the supported node hardware the safe and the unsafe method. Unsafe updates cannot guarantee that a node reboots in a sane state, while safe updates have a fallback mechanism which uses the old image in case the new one is faulty or not bootable.

6.6.1 Safe Update with GRUB

If a node platform can use GRUB and has a block device as secondary storage, which can be partitioned and is big enough for storing two software images, the special safe update mechanism for software images can be used. Figure 6.6 shows the four possible states of the boot block and the GRUB configuration file on the block device of the node. The GRUB boot loader is able to rewrite the MBR at boot time to boot another partition the next time the system is started. GRUB is configured by a configuration file on each bootable partition, which is accessible and changeable through the operating system. These two features enable the implementation of the safe update mechanism, which includes three main procedures. These three procedures default operation, successful update and faulty update get explained in more detail with help of the four states (a) through (d), shown in Figure 6.6.

- Default operation
  (a)→reboot→(a)

  The boot block is setup as depicted in state (a). The default image is booted. When no changes are made to GRUB configuration files, the boot block is still in state (a) and the default image is loaded again at the next reboot.

- Successful update
  (a)→initialise update→(b)→move update to default→(c)→reboot→(a)

  The boot block is setup as depicted in state (a). An update is initialised by copying the update image to the first partition and adjusting the GRUB configuration file on the first partition to boot *entry 1*. As soon as the system is rebooted the node is in state (b). Therefore the update image is booted. After successful booting of the update image, the default image is replaced with the update image and the GRUB configuration file on the first partition is readjusted to boot *entry 0*. After a reboot the node resides in state (c) and the update image is installed as default. Rebooting once more causes the node to stay permanently in state (a) which is default operation again.

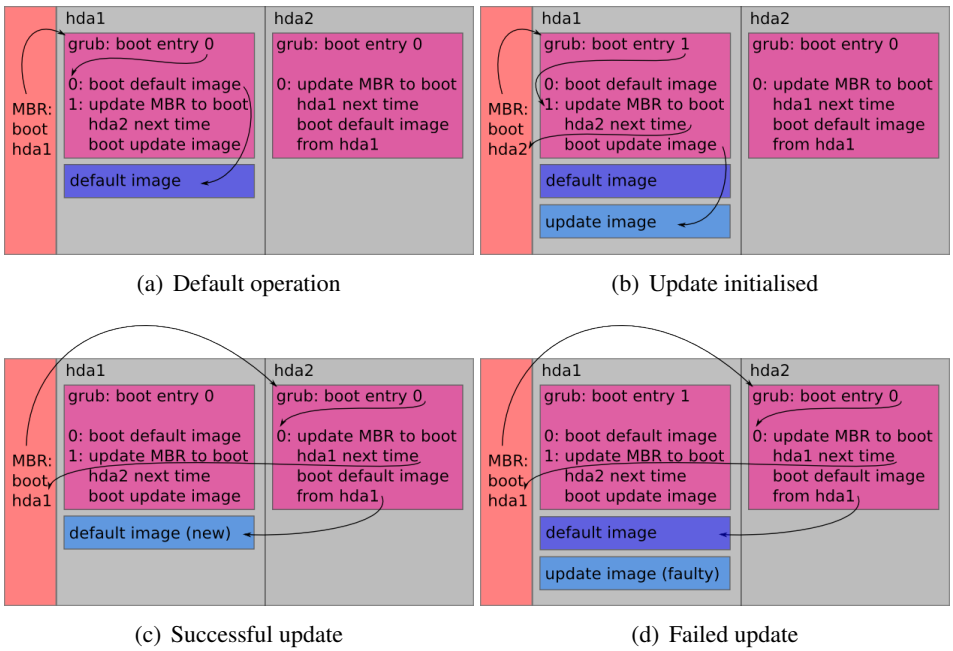


Figure 6.6: Safe update with GRUB

- Failed update
  (a)→initialise update→(b)→automatic reboot→(d)→remove update→(a)

  The boot block is setup as depicted in state (a). An update is initialised by copying the update image to the first partition and adjusting the GRUB configuration file on the first partition to boot *entry 1*. At reboot the node is in state (b). Therefore the update image is booted. An automatic reboot is triggered if the image is corrupt, causes kernel panics or cannot boot at all. Since no adjustments to the GRUB configuration file on the first partition could have been made the node is in state (d) and loads the old default image as fallback. The faulty update image is removed and the GRUB configuration file on the first partition is adjusted to boot *entry 0*. After the next reboot the node is again in state (a) which is default operation.

6.6.2 Unsafe Update with Custom Boot Loader

Nodes which cannot use GRUB or do not have enough secondary storage for two software images do not profit from the safe update mechanism shown in the previous subsection. E.g. the Meraki node does not fulfil any of these conditions. Its boot loader can only boot one image from flash and is not configurable through the operating system. Moreover, the size of the flash storage does not allow using

more than one software image. It further requires the installation of the software image to the flash with the dd program, which takes a very long time due to a block emulation overhead in the kernel. The installation method takes up to 45 minutes and causes a heavy load on the node. For reducing the load while the dd program is running, the configuration engine processes have to be temporary stopped. A wait time of three minutes between the arrival of an new software image and its installation ensures the further propagation of the software image to other nodes before the configuration engine is temporary interrupted and the dd program is started.

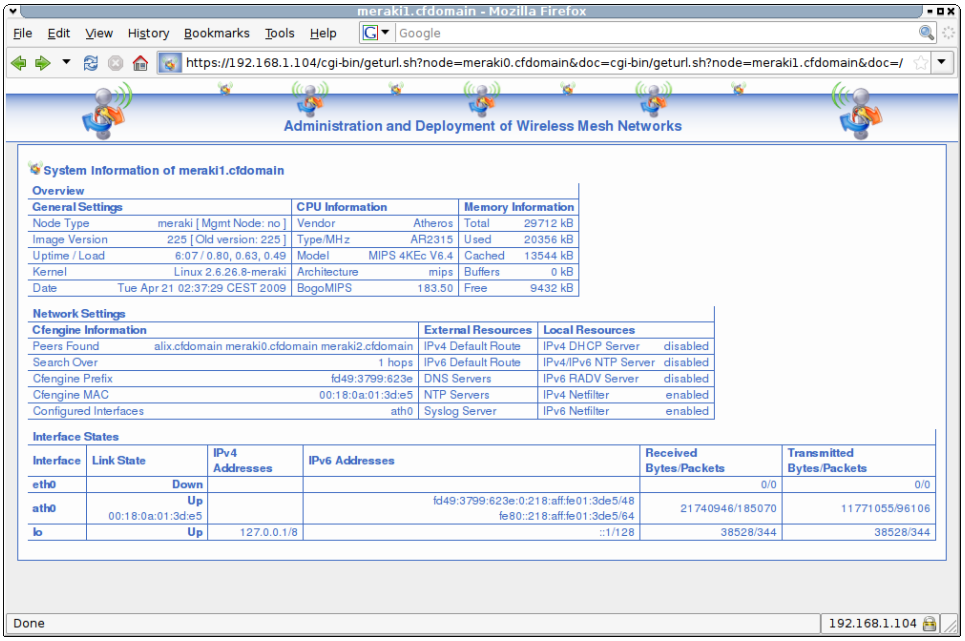
There is also no possibility for an automatic recovery to an old working image once a faulty Meraki software image was distributed in the network and has been installed on a node. Physical access to the nodes would be needed for the reinstallation of a working software image. Therefore, it is advisable to check new software images for Meraki nodes on a well accessible test node before distributing them with the image update module. The software image installation method used on Meraki nodes is therefore intentionally called unsafe.

6.7 Node System Information Web Interface

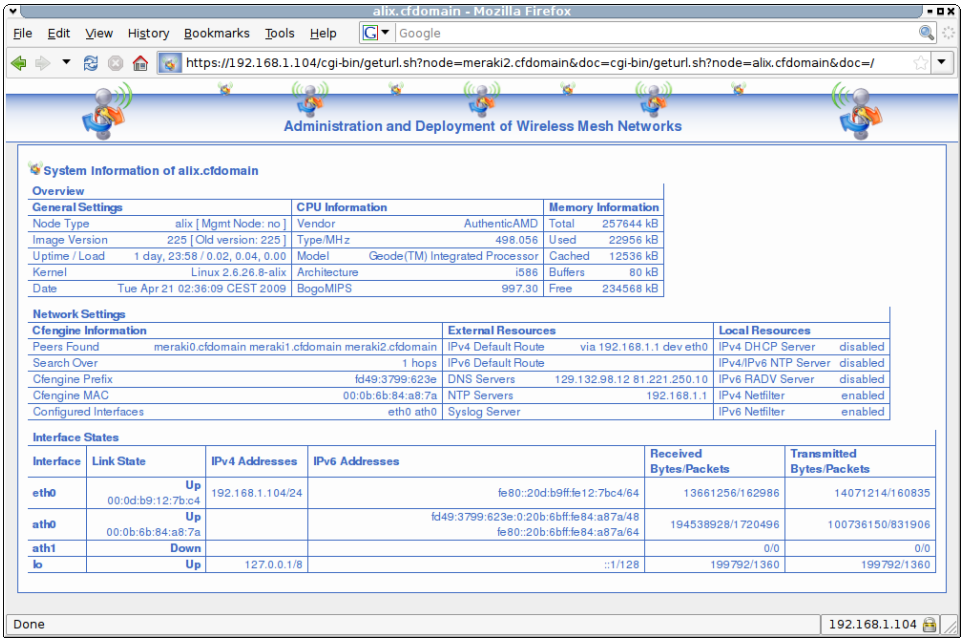
Since an IPv6 and SSL capable web server is already installed and used for system clock synchronisation between nodes, a little system information CGI script is available on the nodes. This CGI script provides statistics and crucial node system information, like reachable peers or actual network settings in a read-only web page. Furthermore the *geturl.sh* CGI script is provided, which can display a defined web document from an other node. Using *geturl.sh* recursively allows an administrator to view important configuration parameters of all nodes fast, needing only access to one node's web interface. One drawback is that the current path through the network has to be provided by the administrator. Since the system information CGI script provides information on reachable peers, this path can also be calculated recursively by the administrator. An example of such a recursive usage of the *geturl.sh* CGI script and the resulting web document is shown in Figure 6.7.

6.8 Sample Network Configuration File

For completeness, a sample network configuration file for an Alix node (*network.conf*) is listed here to show what configuration parameters are exactly supported by the network configuration module.



(a) System information of meraki1



(b) System information of alix

Figure 6.7: Node system information web interface

```
# /etc/conf.d/network.conf
# Each node needs this file and and copy of it in
# /etc/network.d, which holds the network.confs of all nodes.

#########################################################
# Static section: this gets set __once__ by the GUI.
# Changes in this section should generally not be made.
#########################################################

# System type, this determines the image type.
# Supported are: alix wrap and meraki.
GLOBAL_SYSTYPE="alix"

# Available hardware devices on platform.
# Use the device names the kernel sees.
AVAIL_DEVICES="eth0 ath0 ath1"

# Which devices are WIFI capaple.
GLOBAL_WIFI_DEVICES="ath0 ath1"

# Update type, this determines what firmware update
# mechanism is used. Supported are: none, safe, unsafe.
GLOBAL_UPDATE_TYPE="safe"

# If this particular node is a mgmt node.
GLOBAL_MGMT="no"

# REQUIRED !!!
# Cfengine ipv6 configuration net prefix.
GLOBAL_CFNET_PREFIX="fd49:3799:623e"

# REQUIRED !!!
# The MAC address of one interface in GLOBAL_WIFI_DEVICES.
GLOBAL_CFNET_MAC="01:02:03:04:05:06"

#########################################################
# Dynamic section:
# All values below here can be configured as wished.
#########################################################

# Over how many hops the node should search
# for cfengine peers.
GLOBAL_CFHOPS="1"

# When a node has seen no peers in the last GLOBAL_LOSTTIMEOUT
# hours, it regards itself as misconfigured.
GLOBAL_LOSTTIMEOUT="24"

# Devices to setup by /etc/init.d/rc.network.
# Only devices listed here get setup by initscripts.
GLOBAL_DEVICES="eth0 ath0"

# Static routing.
GLOBAL_GW6=""
GLOBAL_GW6_IF=""
GLOBAL_GW=""
GLOBAL_GW_IF=""

# DNS servers.
GLOBAL_DNS0=""
GLOBAL_DNS1=""
```

```
# ADHOC routing.
GLOBAL_ADHOCROUTING="olsr"
GLOBAL_ADHOCROUTING_DEVICES=""

# Network time protocol.
# Is this node an ntp server?
GLOBAL_NTP_BE_SERVER="no"
# Is this node an ntp client?
GLOBAL_NTP_BE_CLIENT="no"
# If it is client, use a server pool to get the time.
GLOBAL_NTP_POOL="no"
# If it is client, use this server or server pool.
GLOBAL_NTP_SERVER=""

# IPv6 router advertisement daemon.
GLOBAL_RADVD_START="no"

# IPv4 DHCP server.
# Start a DHCP server?
GLOBAL_UDHCPD_START="no"
# Only one device is supported at the moment.
GLOBAL_UDHCPD_DEVICE=""
# Start IP of the leases pool.
GLOBAL_UDHCPD_LEASE_START=""
# End IP of the leases pool.
GLOBAL_UDHCPD_LEASE_END=""

# Syslog server.
# Only IPv4 adresses supported by busybox.
GLOBAL_SYSLOG=""

# Netfilter section.
# You can use both Ipv4 and/or IPv6 netfilter.
# IP forwarding (4 and 6) is disabled by default in /etc/sysctl.conf.
# Enable logging of all dropped pakets in IPv6 and IPv4 filters?
NETFILTER_LOG="yes"
# Start netfilter IPv4?
NETFILTER_START="no"
# IPv4 input.
NETFILTER_ALLOW_LOCAL_IN_TCP="22 443"
NETFILTER_ALLOW_LOCAL_IN_UDP="67 123"
# IPv4 output.
NETFILTER_ALLOW_LOCAL_OUT="yes"
# IPv4 forward.
NETFILTER_ALLOW_FORWARD="yes"
# Device for SNAT.
NETFILTER_SNAT_TO_IF=""
# Start netfilter IPv6?
NETFILTER6_START="no"
# IPv6 input (https cfservd tracroute6 are needed).
NETFILTER6_ALLOW_LOCAL_IN_TCP="22 443 5308"
NETFILTER6_ALLOW_LOCAL_IN_UDP="123 33434:33999"
# IPv6 output.
NETFILTER6_ALLOW_LOCAL_OUT="yes"
# IPv6 forward.
NETFILTER6_ALLOW_FORWARD="yes"

############################################################
# Network device section.
# Each device in GLOBAL_DEVICES needs a device section.
############################################################
```

```
# Device eth0.
# Use a DHCP client for v4 setup.
eth0_DHCP="no"
# Let DHCP overide GLOBAL_GW.
eth0_OVERRIDE_ROUTE="yes"
# Let DHCP overide GLOBAL_DNS.
eth0_OVERRIDE_DNS="yes"
# Let DHCP overide GLOBAL_NTP_SERVER.
eth0_OVERRIDE_NTP="yes"
# Static IPv4 address and netmask.
eth0_IP=""
eth0_MASK=""
# IPv4 domain for static resolving.
eth0_DOMAIN=""
# Static IPv6 address and netmask.
eth0_IP6=""
eth0_MASK6=""
# IPvv6 domain for static resolving.
eth0_DOMAIN6=""
# IPv6 radvd prefix.
eth0_RADVD_PREFIX=""

# Device ath0.
ath0_DHCP="no"
ath0_OVERRIDE_ROUTE="no"
ath0_OVERRIDE_DNS="no"
ath0_OVERRIDE_NTP="no"
ath0_IP=""
ath0_MASK=""
ath0_DOMAIN=""
ath0_IP6=""
ath0_MASK6=""
ath0_DOMAIN6=""
ath0_RADVD_PREFIX=""
# 802.11 standard a,b,g,n or auto.
ath0_STANDARD="g"
# 802.11 essid.
ath0_ESSID=""
# Positive integer in mW, auto means adjust automatically.
ath0_TXPOWER="auto"
# Channel integer from 1 to 11
ath0_CHANNEL="1"

# Device ath1.
ath1_DHCP="no"
ath1_OVERRIDE_ROUTE="no"
ath1_OVERRIDE_DNS="no"
ath1_OVERRIDE_NTP="no"
ath1_IP=""
ath1_MASK=""
ath1_DOMAIN=""
ath1_IP6=""
ath1_MASK6=""
ath1_DOMAIN6=""
ath1_RADVD_PREFIX=""
ath1_STANDARD="g"
ath1_ESSID=""
ath1_TXPOWER="auto"
ath1_CHANNEL="5"
```

Listing 6.4: Sample network configuration file (*network.conf*)

Chapter 7

Evaluation

The evaluation of ADAM framework is divided in two parts. First the ADAM build system is evaluated using the three supported platforms. The second part consists of a detailed evaluation test procedure for the ADAM configuration framework.

7.1 Evaluation of the ADAM Build System

This section focuses on the evaluation of the ADAM build system for the three supported platforms, and includes the evaluation of image creation and deployment procedures. Moreover, the extensibility of the ADAM build system is evaluated by adding support for new packages and target platforms.

7.1.1 Build System

The evaluation of the build system is a difficult task since every cross compiled binary or library has to be tested. These binaries or libraries have to be tested on all three involved platforms. Therefore only the minimal package selection for the devices is built and not every single binary is tested. All the functionality needed for a working configuration framework is crucial and therefore the test procedures outlined in Section 7.2 are regarded as sufficient tests for the cross compiled software. The test procedures outlined use the most installed cross compiled software anyway, a defect in an important software component would be detected. The evaluation tests which can be made for testing the build system include therefore a cross compilation of the default software selection for all three involved hardware platforms with *build-tool*. If *build-tool* successfully returns from the target board setup, toolchain installation and package build procedures for all three platforms the test can be regarded as successful.

7.1.2 Image Creation

Evaluating the image creation process consists mainly of manually inspecting the resulting images. Therefore configuration images can be mounted as loopback

Package	Version	Purpose
Hostapd	0.6.4	Tools for a wireless access point
Wpa-supplicant	0.6.4	Tools for a wireless client
PHP	5.2.6	PHP Hypertext Parser
Lighttpd	1.4.20	Lightweight web server with PHP support
Ipsec-tools	0.7	Linux IPSec utilities
Hotplug2	0.9	Removable device management

Table 7.1: Additional packages

file on any Linux system. Software images are in binary form, therefore only the embedded initramfs archive can be inspected by unpacking it with the cpio program. First, the image creation framework is used to generate software and standalone images for all three involved platforms. Second, configuration images for the test network described in Chapter 7.2 are created. In a third step the the initial network configuration also described in Chapter 7.2 is injected. After *build-tool* has successfully returned from all these procedures the manual inspection of the images is done as described above.

7.1.3 Deployment

The hardware deployment process is evaluated as follows. Deployment of the resulting images on the involved nodes is achieved according to the available deployment documentation in README.*board*. This is heavily depending on the node hardware and each supported node has its own deployment documentation. Deployment is exactly done as described in these README files.

7.1.4 Additional Packages

To evaluate the extensibility of the ADAM build system, it is crucial to test the support for building additional software packages. Additional packages have been successfully built with ADAM for the three supported platforms. An incomplete list of these additional packages is shown in Table 7.1.

7.2 Evaluation of the ADAM Configuration Framework

For the evaluation of the ADAM configuration framework a test network is needed. The network used for the evaluation tests consists of four nodes, one Alix and three Meraki nodes (e.g. *alix*, *meraki0*, *meraki1* and *meraki2*). In the default setup all nodes are located in a way, that they are able to reach any other node (e.g. in one

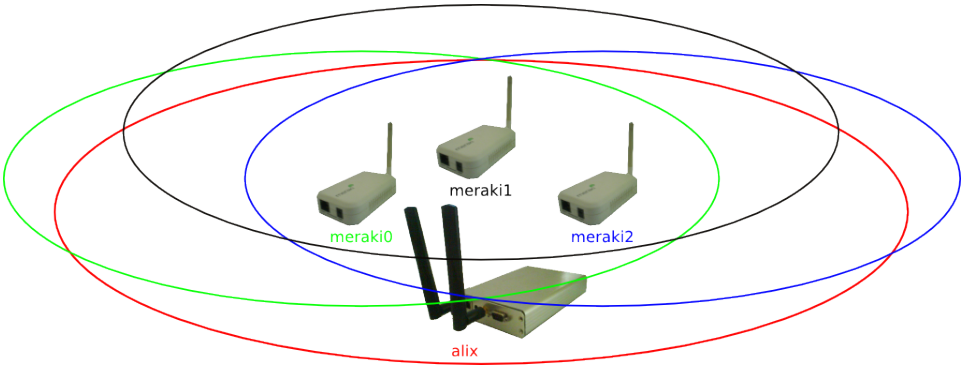


Figure 7.1: Default setup of the test network

room). This default setup of the test network is shown in Figure 7.1. All evaluation tests were made with subversion 225 of the ADAM configuration framework.

7.2.1 Setup

The initial network configuration for the four nodes is created manually by the administrator and looks the same for all nodes, except for the MAC address of the wireless interface used by Cfengine and platform specific settings like node or update type. The initial network configuration used is minimal, neither static IPv4 or IPv6 addresses nor routing is configured for the nodes. This allows evaluating the dedicated IPv6 network used for cfagent and cfservd connections. Moreover, the Alix node is additionally configured to set up its wired network interface by DHCP. This allows the administrator to login on the *alix* node with an SSH client. First, only two Meraki nodes (*meraki0* and *meraki1*) are deployed, *meraki2* is kept back for testing the new node module later.

The *alix* node is booted first. If connected to a network, which provides Internet access, the *alix* node tries to synchronise its system clock with an external NTP server. If no Internet access is available, the administrator can set the time also manually on the *alix* node with the date command. Setting the system clock is optional, the configuration framework works without it. Nevertheless it is useful to have a reasonably accurate time on the nodes. Once the alix node is up, the administrator logs in with an SSH client or over a serial console and is able to read the system log files. Moreover, the administrator can use the system information web interface on the *alix* node to inspect crucial system information.

If logged into the *alix* node, the administrator starts the different test procedures. The following subsections describe these different test procedures and Table 7.2 shows their duration on the different nodes, respectively.

7.2.2 Peer Detection

Evaluating the peer detection mechanism used by the ADAM configuration framework works as follows. The not yet booted nodes (*meraki0*, *meraki1*) are powered on by the administrator. After the boot procedure of the two nodes has been completed, and each node's cfagent has been started at least once, each node has two peers. This can be verified using the system information web interface on the *alix* node or by logging into the nodes. The actual calculated peers are listed in the */tmp/cfpeers* file on any node or shown directly in their web interface. Table 7.2 shows the boot times (from power on to login prompt) and the duration from the end of the boot process until the node has found the two peers.

7.2.3 System Clock Synchronisation

Evaluating the system clock synchronisation mechanism is easy. Being already peers of each other, all nodes have their system clock therefore already synchronised. The system clock on any node in the network can be evaluated by issuing the date command. For further evaluation of the system clock synchronisation a Meraki node is rebooted. After the first run of cfagent on the rebooted node, the system clock is set to the correct time again. This can be verified also by inspecting the system log file on the node. Table 7.2 shows the duration from the end of the boot process until the system clock is synchronised.

7.2.4 Misconfigured Nodes

The proper functionality of the detection mechanism for misconfigured nodes is evaluated as follows. A randomly selected node in the network is configured with a too low transmission power for reaching other nodes. As configured, one day later the node has rebooted and the transmission power settings have been restored as described in Subsection 6.3.6. In addition the location of a randomly selected node is changed, until no other node is reachable. As expected, on the node the mechanism for misconfigured nodes is also triggered, as long as it cannot reach any other the node it is rebooted every day. This procedure takes 24 hours as defined in the default configuration. This value is adjustable in every node's *network.conf* file and can be set to one hour for a faster test procedure. Table 7.2 shows the duration from the setting of transmission power until the automatic re-appearance of the node.

7.2.5 Network Configuration Module

For testing the network configuration module, static IPv4 and static IPv6 addresses as well as static host name resolving is configured on the nodes. Therefore all network configuration files for the nodes in */etc/network.d* have to be adjusted. The adjustments can be done on any arbitrary selected node. The successful distribution of the new configurations, can be verified by pinging all configured IPv4 and

IPv6 or host names respectively. Table 7.2 shows the duration from deploying the new configuration until the node's newly configured address is reachable.

7.2.6 New Node Module

The kept back node *meraki2* is now deployed as new node and powered on afterwords. After deployment and a completed boot process *meraki2* has already all keys and *network.conf* files of the other nodes. Therefore it has already valid peers, which can be verified by inspecting its */tmp/cfpeers* file. Since the other nodes have neither a key nor a network configuration file of *meraki2*, it will not be contained in their */tmp/cfpeers* file. To distribute the new key and the network configuration file, the new key *root-meraki2.cfdomain.pub* is copied to */var/cfengine/newkeys* and the new network configuration file *meraki2.conf* to */var/cfengine/newkeys*. This can be done on any of the other nodes (*meraki0*, *meraki1* and *alix*). The successful integration is verified by inspecting the */tmp/cfpeers* found on *meraki0*, *meraki1* and *alix*, respectively. Table 7.2 shows the duration from deploying new the key and configuration file until the new node is in the list of current peers.

7.2.7 System Update Module

The system update module is tested by updating the *alix* node with a new working software image first. Second, software images of the Meraki nodes are updated. In a third step the *alix* node is updated with a faulty software image to verify the safe update mechanism. Therefore new working software images for both the Meraki and the Alix platform have been generated with the ADAM build system. A faulty image for the alix node is made by creating a file from */dev/zero*, 5 MByte in size. The images have to be distributed in a serial way. The working *alix* image has to be copied to */var/cfengine/update* for being distributed. After the the update on all nodes of the corresponding type has been completed, the image is replaced with a file, which size is zero. Then the working *meraki* image can be copied to */var/cfengine/update* to be distributed. The procedure is repeated with the faulty image for the *alix* node, after all Meraki nodes have been updated. Table 7.2 shows the duration from deploying the new image until the node is up again.

7.2.8 Command Module

The command definition file shown in Listing 6.1 is taken to test the functionality of the command module. The script defined in this command definition file copies a sample init script and a sample configuration file to all Alix nodes and one dedicated Meraki node (*meraki0*) as shown in Listing 6.2. The command definition file is now altered, that the script is executed on all nodes, and copied to the */var/cfengine/commands* directory together with the script and the sample files. The proper execution of the script is then evaluated by inspecting the corresponding node reply file. Table 7.2 shows the duration from deploying the command

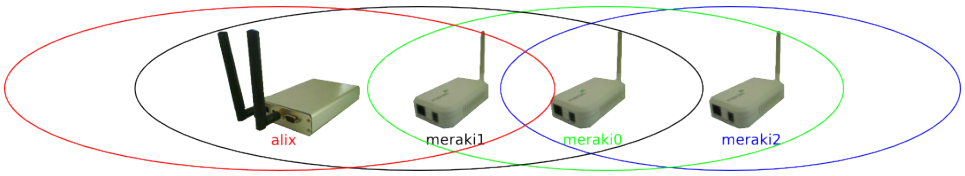


Figure 7.2: Linear setup of the test network

until the node executes it and the duration from deploying the command until the node's reply file is available.

7.2.9 Distribution over Multiple Hops

To evaluate distribution over multiple hops the topology of the test network has to be adjusted. Therefore the four nodes are aligned in a chain (*alix*, *meraki0*, *meraki1*, *meraki2*), that one node has only the direct neighbour in the chain as peer. This linear setup of the test network is shown in Figure 7.2. With this setup the evaluation tests for the command module are repeated to measure the configuration distribution from one end of the chain (*alix*) to the other (*meraki2*). Table 7.2 shows the duration from deploying the command until the node executes it and the durations from deploying the command until the node's reply file is available.

Node	*alix*	*meraki0*	*meraki1*	*meraki2*
Core architecture				
Image loaded by boot loader	2 sec	3:57 min	3:59 min	n/a
Boot procedure (first time)	32 sec	5:08 min	5:11 min	n/a
Boot procedure (default)	22 sec	4:20 min	4:18 min	n/a
Two peers found	1:38 min	1:00 min	1:26 min	n/a
System clock synchronised	1:41 min	1:01 min	1:27 min	n/a
Misconfigured node up again	24 hours	n/a	24 hours	n/a
Network configuration module				
Reply from new address	23 sec	1:53 min	1:20 min	n/a
New node module				
New node (*meraki2*) found	1:30 min	56 sec	1:47 min	n/a
System update module				
Reachable after successful update	59 sec	50:34min	51:02 min	50:40 min
Reachable after faulty update	45 sec	n/a	n/a	n/a
Command module				
Command executed	50 sec	1:20 sec	21 sec	1:40 min
Reply file received on *alix*	50 sec	2:56 min	2:56 min	2:56 min
Distribution over multiple hops				
Command executed	1:02 min	41 sec	3:36 min	4:54 min
Reply file received on *alix*	1:02 min	1:06 min	4:32 min	6:46 min

Table 7.2: Evaluated durations for the different test procedures

Chapter 8

Conclusion and Future Work

This chapter tries to concentrate the experiences made while developing and testing the ADAM software framework.

8.1 Conclusion

The new software suite is a user-friendly, easy to understand, simply extensible framework to build customised Linux software images for embedded devices with limited storage capacities like for example nodes of a wireless mesh network. The complex and time-consuming process from the setup for a specific target platform over creating a cross compilation toolchain and compiling software packages to image creation is done with two easy to handle front-ends *build-tool* and *image-tool*. The prerequisites for using the software suite for the system used as host platform are minimal. The new implementation is designed to support a growing number of target devices and is therefore easily extensible to other embedded system hardware. In addition it provides a configuration framework for a resulting heterogeneous wireless mesh network consisting of the different node hardware platforms.

8.1.1 ADAM Build System

The ADAM build system handled by *build-tool* is clearly designed and heavily tested with many platforms and software packages. Being as flexible as possible regarding packages selection and possible target platforms as well as the minimal prerequisites for the host platform are major advantages of the ADAM build system. Currently, the ADAM build system meets all requirements to support the successful deployment and operation of WMN. The build system is very powerful and many packages can be built for many different hardware platforms. Nevertheless, many new features and enhancements are imaginable for an even better build system.

8.1.2 Creation of ADAM Images

The creation of the different images handled by *image-tool* is very simple, its structure is designed with the all-in-one software image in mind. The image creation process satisfies the needs for small embedded devices and a manageable amount of packages. If e.g. a package manager is used the image creation process has to be adapted and requires a design change. However, more software can be installed on the nodes with the all-in-one software image, than with any package manager based solution. The idea of a per device configuration image separated from the actual software image works well and has clear advantages for the update and configuration procedures. Software images can be updated much faster since they are the same for one node type.

8.1.3 ADAM Configuration Framework

The integration of the configuration framework based on Cfengine in the build system and image creation process worked well, and it was the right decision to write a new build system for this purpose. Cfengine can serve as a basic configuration system for wireless mesh nodes, which can easily be adjusted to future needs. The use of Cfengine only has disadvantages when really small embedded devices are targeted. Devices smaller than the Meraki Mini (32 MByte RAM and 8 MByte flash storage) are not in the focus of the ADAM framework. The new implementation of the ADAM configuration framework has clear advantages over the one proposed by SRM, as it removes the dependencies from network configurations.

8.2 Future Work

Any suggestions for enhancing the ADAM framework are very welcome. The ADAM framework is used in several projects of the research group Computer Networks and Distributed Systems (CNDS) at the Institute of Computer Science and Applied Mathematics of the University of Bern. The author and the CNDS research group provide the software in a public subversion repository available at *https://guest:guest@subversion.cnds.unibe.ch/svn/adam*.

Index of Acronyms

ADAM Administration and Deployment of Adhoc Mesh, 1, 3, 4, 6, 10, 12, 13, 17–21, 25, 26, 28, 31, 32, 40, 43, 44, 49, 54, 57–59, 61, 62, 66, 72, 81–83, 85, 89, 90
AODV Adhoc On-Demand Vector Routing, 19–21
ARM Advanced RISC Machine, 4
ATMA Framework for the Management of Large-Scale Wireless Network Testbeds, 20, 21

Bash Bourne Again SHell, 18, 22, 32, 33, 58

CGI Common Gateway Interface, 62, 76
CLFS Cross Linux From Scratch, 19
CNDS Computer Networks and Distributed Systems, 90
CPU Central Processing Unit, 3, 12

DAMON Distributed Architecture for Monitoring Multi-hop Mobile Networks, 20, 21
DHCP Dynamic Host Configuration Protocol, 23, 45, 54, 57, 67, 69, 83
DNS Domain Name System, 54, 57, 69

ELF Executable and Linkable Format, 11
ESSID Extended Service Set IDentifier, 62
Ext2 Second Extended Filesystem, 45, 48, 51

GCC GNU Compiler Collection, 12
Glibc GNU C library, 12
GNU GNU is Not Unix, 9, 11
GRUB GRand Unified Bootloader, 49, 51, 74, 75
GSM Global System for Mobile communications, 2
GUI Graphical User Interface, 1, 19, 49, 57, 66, 70

IEEE Institute of Electrical and Electronics Engineers, 2
IETF Internet Engineering Task Force, 61
IPv4 Internet Protocol Version 4, 19, 20, 54, 58, 67, 69, 83, 84
IPv6 Internet Protocol Version 6, 23, 54, 57–59, 61, 62, 65, 66, 69, 76, 83, 84

JANUS Framework for Distributed Management of Wireless Mesh Networks, 20, 21
JFFS2 Journalling Flash File System version 2, 45, 51

LFS Linux From Scratch, 19

MAC Media Access Control, 54, 58, 61, 83
MAYA Tool For Wireless Mesh Networks Management, 19–21
MBR Master Boot Record, 51, 74
MCL Mesh Connectivity Layer, 21
MIPS Microprocessor without Interlocked Pipeline Stages, 4, 42
MTD Memory Technology Device, 51

NAT Network Address Translation, 45, 69
NFS Network File System, 45
NTP Network Time Protocol, 22, 23, 54, 57, 62, 69, 83

PHP PHP Hypertext Parser, 14
PID Process IDentfier, 52

RAM Random-Access Memory, 3, 6, 9, 12, 13, 18, 44, 45, 48, 49, 52, 70, 90
RFC Request For Comment, 58, 61

SNMP Simple Network Management Protocol, 20
SoC System on a Chip, 3
SRM Secure Remote Management and Software Distribution for Wireless Mesh Networks, 17, 22, 23, 57, 58, 62, 90
SSH Secure SHell, 52, 83
SSL Secure Sockets Layer, 52, 59, 62, 76

UDP User Datagram Protocol, 19

WMN Wireless Mesh Network, 1–3, 6, 9, 21, 25, 45, 89
WMNs Wireless Mesh Networks, 1–3, 6, 9, 10, 17, 19–22, 58
WRAP Wireless Router Application Platform, 3, 4, 22, 51

Bibliography

[1] I. F. Akyildiz and X. Wang, "A survey on wireless mesh networks," *Communications Magazine, IEEE*, vol. 43, no. 9, pp. S23–S30, 2005.

[2] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless mesh networks: a survey," *Computer Networks Journal (Elsevier)*, vol. 47, no. 4, pp. 445–487, 15 March 2005.

[3] R. Bruno, M. Conti, and E. Gregori, "Mesh networks: commodity multihop ad hoc networks," *IEEE Communications Magazine*, vol. 43, no. 3, pp. 123–131, March 2005.

[4] M. L. Sichitiu, "Wireless mesh networks: Opportunities and challenges," in *Wireless World Congress*, Palo Alto, California, USA, May 2005.

[5] Meraki Networks, Inc., "The Meraki Mini / Indoor Wireless Platform," http://meraki.com, 2007.

[6] PC Engines GmbH , "Alix Platform (Alix)," http://www.pcengines.ch, 2007.

[7] PC Engines GmbH, "Wireless Router Application Platform (WRAP)," http://www.pcengines.ch, 2006.

[8] Openmoko, Inc., "The NEO Freerunner," http://openmoko.com/product.html, 2008.

[9] Ikanos Communications, Inc., "Fusiv Vx180 Highly Integrated VDSL Gateway Processor," http://www.ikanos.com/products/vdsl/cpe-chipsets/fusiv-vx180/, 2008.

[10] L. Torvalds, "The Linux 2.6 Kernel," http://kernel.org, 2004.

[11] GCC Steering Comitee, "GNU Compiler Collection (GCC)," http://gcc.gnu.org, 2007.

[12] Roland McGrath et al., "GNU C Library stable release (Glibc)," http://www.gnu.org/software/libc, 2007.

[13] E. Andersen, "uClibc," http://www.ulibc.org, 2006.

[14] R. Landley, “Busybox,” http://www.busybox.net, 2006.

[15] M. Burgess, “Cfengine: a system configuration engine,” http://www.cfengine.org, 1993.

[16] T. Staub, D. Balsiger, M. Lustenberger, and T. Braun, “Secure Remote Management and Software Distribution for Wireless Mesh Networks,” in *7th International Workshop on Applications and Services in Wireless Networks (ASWN 2007)*, Santander, Spain, May 2007, pp. 47–54.

[17] M. Baker, G. Rozema, I. Kaloz, N. Thill, F. Fainelli, F. Fietkau, M. Albon, and T. Yardley, “OpenWrt,” http://openwrt.org, 2006.

[18] H. H. P. Freyther, K. Kooi, D. Vollmann, J. Lenehan, M. Juszkiewicz, and R. Leggewie, “Openembedded,” http://openembedded.org, 2006.

[19] G. Beekmans, “Linux From Scratch (LFS),” http://www.linuxfromscratch.org, 2006.

[20] D. Manzano, J.-C. Cano, C. Calafate, and P. Manzoni, “Maya: A tool for wireless mesh networks management,” *Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE Internatonal Conference*, pp. 1–6, Oct. 2007.

[21] C. Perkins, E. Belding-Royer, and S. Das, “Ad hoc On-Demand Distance Vector (AODV) Routing,” IETF RFC 3561, July 2003.

[22] K. N. Ramachandran, K. C.Almeroth, and E. M. Belding-Royer, “A framework for the management of large-scale wireless network testbeds,” in *1st Workshop on Wireless Network Measurements (WiNMee 2005)*, Riva del Garda, Trentino, Italy, April 3 2005.

[23] K. N. Ramachandran, E. M. Belding-Royer, and K. C. AImeroth, “Damon: a distributed architecture for monitoring multi-hop mobile networks,” in *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (IEEE SECON 2004)*, Santa Clara, CA, USA, October 4 - 7 2004, pp. 601–609.

[24] R. Riggio, N. Scalabrino, D. Miorandi, and I. Chlamtac, “JANUS: A framework for distributed management of wireless mesh networks,” *Testbeds and Research Infrastructure for the Development of Networks and Communities, 2007. TridentCom 2007. 3rd International Conference on*, pp. 1–7, May 2007.

[25] Microsoft, “Mesh Connectivity Layer (MCL),” http://research.microsoft.com/mesh/, 2009.

[26] Yoshinori K. Okuji, “GNU GRand Unified Bootloader (GRUB),” http://www.gnu.org/software/grub/, 1999.

[27] R. Hinden and B. Haberman, “Unique Local IPv6 Unicast Addresses,” IETF RFC 4193, October 2005.