



Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Diplomarbeit

Thema: Development of a Software Distribution Platform

eingereicht von: Bernhard M. Wiedemann

Betreuer: Prof. Jens-Peter Redlich

Beginn: 2005-05-31

Abgabe: 2005-11-30

Berlin, den November 14, 2005

Acknowledgements

Writing this thesis has been amazingly enjoyable for me. For this I have to thank the people behind BRN: Jens-Peter Redlich, Mathias Jeschke and Anatolij Zubow. Thanks for raising my interest in this topic, for inspiring me in the work and for all the motivation you gave. I did not count how many weird bugs I encountered, which you helped me fix.

I also want to thank my colleagues at the job who missed me in the recent months and had to take over part of my workload.

And last, but not least I also got to thank my family for their patience and understanding to let me research and write so many hours without having time to play with my son or to discuss philosophy with my wonderful beloved wife.

Contents

| | |
|---|-----------|
| Contents | V |
| 1 Introduction | 3 |
| 1.1 Thesis | 3 |
| 1.2 Motivation | 3 |
| 1.3 Notation | 4 |
| 1.4 Terminology | 4 |
| 1.5 Thesis Structure | 6 |
| 1.6 Berlin Roof Net | 6 |
| 1.6.1 General Ideas | 6 |
| 1.6.2 Concepts | 8 |
| 2 Problem Statement | 11 |
| 3 Current State of the Art | 13 |
| 3.1 Standards | 16 |
| 3.2 SUSE Linux Update | 16 |
| 3.3 Other Methods | 20 |
| 3.3.1 Package Formats | 20 |
| 3.3.2 Embedded Device Firmware Upgrades | 22 |
| 3.3.3 Current Mesh Network Updates | 23 |
| 4 The Software Distribution Platform | 25 |
| 4.1 Requirements | 25 |
| 4.2 SDP Design | 26 |
| 4.2.1 Using current methods | 26 |
| 4.2.2 Robustness | 30 |

| | | |
|----------|---|-----------|
| 4.2.3 | Notification | 31 |
| 4.2.4 | File Transfer | 31 |
| 4.2.5 | Versioning | 31 |
| 4.2.6 | Consistency of versions | 32 |
| 4.2.7 | Security | 33 |
| 5 | SDP Implementation | 37 |
| 5.1 | Click Modular Router | 37 |
| 5.2 | Time Synchronization | 40 |
| 5.3 | Scheduling Updates | 44 |
| 5.3.1 | Test Versions | 44 |
| 5.4 | Software Package | 45 |
| 5.4.1 | TFTP | 46 |
| 5.4.2 | Managing versions | 46 |
| 5.4.3 | Storage Considerations | 47 |
| 5.5 | Analyzing the Code | 48 |
| 5.6 | Testing and Debugging | 48 |
| 5.7 | Scenarios | 49 |
| 5.7.1 | Normal Operation | 50 |
| 5.7.2 | Normal Update | 50 |
| 5.7.3 | Update with a Disconnected Node | 51 |
| 5.7.4 | Update with a Disconnected Subnet | 51 |
| 5.8 | Scalability | 51 |
| 5.9 | Effort Estimation | 52 |
| 6 | Conclusions | 53 |
| 6.1 | Summary | 53 |
| 6.2 | Future Work | 53 |
| | Appendices | 54 |
| A | BRN NTP | 55 |
| A.1 | Example UTC Time Sources | 55 |
| A.2 | NTP Implementation | 56 |
| B | Click Configuration Files | 57 |

| | |
|------------------------|-----------|
| <i>CONTENTS</i> | VII |
| List of Figures | 61 |
| Bibliography | 63 |

Abstract

The new Berlin Roof Net (BRN) project aims at providing a decentralized ad-hoc multi-hop wireless mesh network, which is a hot topic of research. A considerable number of multi-hop mesh network routing protocols exist with specific strengths and weaknesses, having parameters and options to optimize throughput, latency, reliability, fairness, security, etc. However, reliably distributing new, improved routing protocols in this decentralized network without centralized resources, unique Internet Protocol address assignment and without routing is quite a challenge. It requires a well designed Software Distribution Platform (SDP). This work will discuss such SDP design and a working implementation in detail.

It will be shown that it is possible to reliably distribute software, using an infection based distribution method, direct links for communication of BRN nodes, TFTP for file transfer, and broadcasts for neighbor notification and time synchronization.

Chapter 1

Introduction

1.1 Thesis

Reliably distributing software within the Berlin Roof Net without routing, unique IP address assignment, or centralized storage while guaranteeing consistent software versions can not be done with existing TCP/UDP/IP software but can be done with specialized software that utilizes a viral distribution method employing local broadcasts for notification of neighboring nodes.

1.2 Motivation

BRN is an ad-hoc multi-hop mesh network, build up from homogenous nodes. This is not a well established technology, but on the contrary, a hot topic of research and development. As is natural with such a new technology, often new requirements arise, and new knowledge is found.

When ideas come up, they are first checked for fundamental flaws and their properties predicted by means of pen and paper. If they are found valid, they may be good and be published, but to find out if the idea could work in real applications and one did not overlook something, one would run a simulated network. If the ideas pass, they may be suitable for practical use, but to be sure one needs to try it on a testbed. Now BRN aims to create such a real world testbed.

However, such a testbed needs supporting infrastructure to easily (e.g. by one click) deploy software to all nodes (amongst other tasks). All BRN development, operation and (especially important for this work) updating is limited by various

factors, including geographic and social problems as well as limited time and monetary resources.

This calls for the creation of a Software Distribution Platform (SDP) that simplifies testing by making it faster and at the same time more consistent and reliable which in turn should result in more acceptance by project participants (the community). It is also desired to build up a community around the BRN to have real users on the testbed so that one can observe its performance under different load and traffic patterns.

To understand how the BRN can benefit from a software distribution platform (SDP), you should read about what BRN is in chapter 1.6.1 on page 6.

It is expected that SDP will speed up development and testing. This should result in a faster, simpler, easier to use, and more comfortable BRN.

1.3 Notation

An URL, file or **code snippet** is written in a monospace font.

Such $\rightarrow term$ is explained in the terminology chapter 1.4.

A citation like [Cohen2003] is written with square brackets.

1.4 Terminology

This defines some heavily used terms and abbreviations

- *WLAN*: Wireless LAN usually denotes all local area networks employing radio waves for transmission. In this work it always refers to networks employing IEEE 802.11g wireless communication protocols.
- *Mesh network*: Is defined as a network that consists of several nodes interconnected with all neighboring nodes, communicating over direct links with each other. In this work links are usually established over WLAN.
- *BRN*: Berlin Roof Net. BRN is a project, initiated by the Systems Architecture Group <http://sar.informatik.hu-berlin.de> aiming to provide a basis for a multi-hop wireless mesh network, to be distributed in Berlin. BRN utilizes the click modular router

software, as designed by MIT's Parallel&Distributed Operating Systems Group <http://pdos.csail.mit.edu/click/> and used in <http://sourceforge.net/projects/roofnet/>

- *SDP*: Software Distribution Platform: a component of BRN managing the distribution of new software to the nodes.
- *CA*: Certification Authority: a trusted party. CA is signing developer or software package certificates. Operation in BRN is based on X509 certificates.
- *DHCP*: Dynamic Host Configuration Protocol: this is used to assign IP addresses to hosts. It is especially useful in conjunction with mobile hosts that travel between different networks (e.g. Laptop, PDA).
- *UTC*: Coordinated Universal Time (former GMT): the time all atomic clocks run on. UTC does not include time zone or daylight saving adjustments.
- *Package*: In this work the word “package” is used to denote a software update package that may consist of several files, that together constitute the working software — on top of the base system.
- *Packet*: A single network packet, usually between 64 and 1500 bytes in size.
- *Broadcast*: For the purpose of this work, this is a local WLAN or Ethernet broadcast packet that is not routed through the mesh network, but received by all neighboring stations.

The following terms are specific to SDP and thus only defined to be used in this work.

- *Version ID*: A plain integer number starting with version 0 for the first public stable release and increasing with each new version. See also chapter 4.2.5 on page 31 about versioning.
- *Meta-info*: This denotes all the information that is stored about a \rightarrow package and transmitted in the meta-info file, including \rightarrow version ID, activation timestamp, as well as names, sizes and hash values of all files in a package. This will be covered later in chapter 5.4 on page 45.

1.5 Thesis Structure

The remainder of this thesis will be split into four major parts.

At first an introduction on BRN will be given.

Next comes a research on the current state of the art which includes analyzing a few methods which are in common use to distribute software today.

Then the design of SDP and its components will be explained.

Finally you will be given insight into the actual code, development process, measurements and other details about the SDP implementation.

1.6 Berlin Roof Net

1.6.1 General Ideas

Because SDP is a part of the BRN project, this work starts by providing general information about BRN and then goes into some details that are important for designing SDP.

BRN is an ad-hoc multi-hop mesh network modeled after the MIT Roofnet in Cambridge [Roofnet]. Ad-Hoc means, that it is possible to establish connectivity without prior knowledge or specific settings like IP numbers, gateways, etc. This allows to employ hardware that can operate out of the box. Consequently ad-hoc networks can be setup much easier than normal networks.

Simple ad-hoc wireless networks exist for some years now, but they require participants to be in direct range with all others. But BRN is a multi-hop wireless network that can relay resources like data or services to neighboring nodes.

A multi-hop mesh network is characterized by several nodes providing services locally, but also being able to relay services to neighboring nodes. In the BRN case communication links are usually established through WLAN.

As mentioned earlier, it is natural with such new, emerging technology that often new requirements arise, and new knowledge comes up. This then requires changes to current protocols and software which in turn makes my Software Distribution Platform essential.

The BRN is a dynamic network of homogenous nodes, that means:

- It has no privileged nodes,
- it runs on homogenous nodes (same hardware and software) with different administrators,

- nodes may be turned off or disconnected for any period of time from the remaining network,
- the network may be split into several sub-networks for any period of time,
- nodes may be added at any time,
- it has no centralized resources (e.g. data-store).

Additionally there have been some design decisions specific to BRN that might change in the future:

- Nodes may be installed on roofs and other locations that are difficult to access,
- it addresses nodes by their unique MAC address and operates at Ethernet level (this is because nodes do not have unique IP addresses assigned),
- nodes act as \rightarrow WLAN access points for IEEE 802.11g standard,
- BRN wants to become a community project/network,
- nodes are setup from inexpensive and thus limited hardware (e.g. WRT54GS has 32MB RAM and 8MB flash).

It is assumed that the BRN project has a trustworthy developer team and CA. This is a necessary assumption, because you can not easily limit what applications can do. And it is even harder to do on limited hardware like the one of BRN nodes.

Node hardware is made up by customary commercial access-points for home and small office usage. Currently it is a Netgear WGT634U, but earlier Linksys WRT54GS was used. It has nearly the same hardware but a different WLAN chip that was not as well documented as the Netgear one. Not surprisingly they contain hardly more hardware than is needed for the job they do: 8MB flash, 32MB RAM, a 200 MHz MIPS CPU and a 802.11g compatible WLAN chip (working with atheros driver).

Why was the BRN started?

People nowadays like having Internet everywhere. Among the possibilities how to access Internet from laptops or handheld devices, it is the wireless technology that is most convenient and easy to use. So what people want, is sit outside

in the garden and surf the web. Laptops and PDAs often have built-in WLAN interfaces complying with IEEE 802.11 b or g standards. However, to let this work, you need a base station somewhere — and it needs a route to the Internet. Of course, everyone can set this up at his home and surf in his own garden, but being able to do so anywhere, and anytime is a new dimension of networking with many new possible applications.

This is the motivational background on which the Berlin Roof Net (BRN) project has set their goal to cover Berlin with a network of BRN nodes that act as WLAN access points.

1.6.2 Concepts

Because SDP is based on BRN and integrated into BRN it is essential to know about some key elements of BRN's design.

To promote the goal of a Berlin-wide network, the project develops and publishes open source software for inexpensive customary hardware. It is intended to not need node configuration of any kind so it can just be installed and deployed to build up a complete wireless network from identical (thus interchangeable) nodes.

So how does it work?

Users can directly contact other BRN users — and also the Internet through gateway nodes. That is possible because from the user perspective BRN looks like a big WLAN access point or Ethernet switch with all users of a wide area (ideally the whole area of Berlin) connected to it. You see this depicted in figure 1.1.

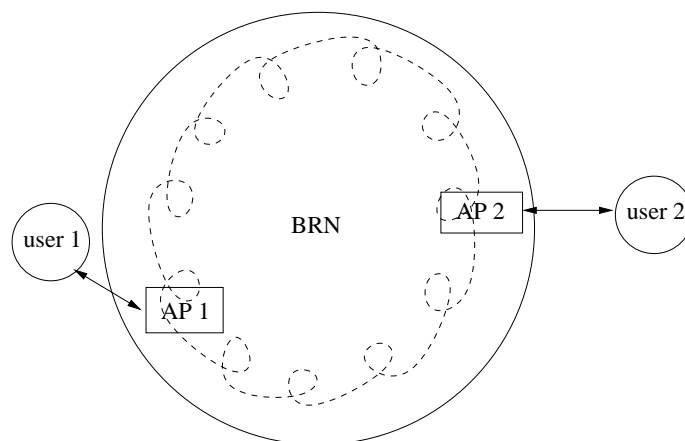


Figure 1.1: BRN from a user's point of view

From the BRN perspective the network receives and forwards Ethernet packets from user machines. This is shown in figure 1.2.

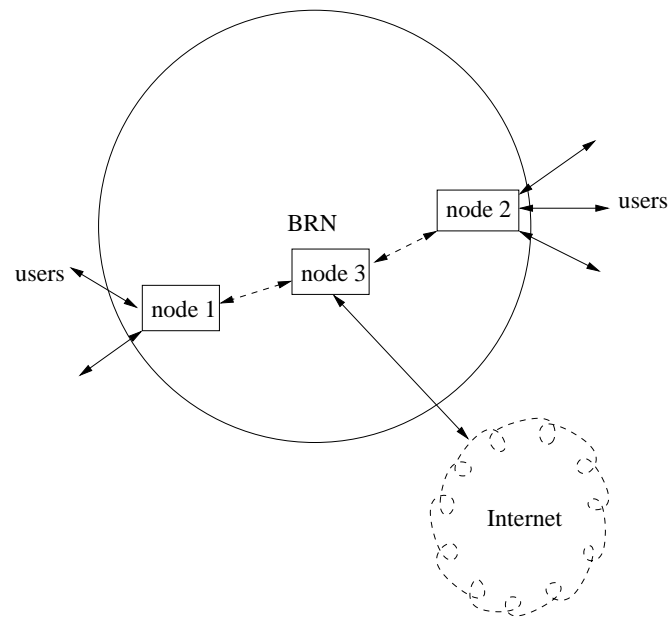


Figure 1.2: BRN from the internal network point of view

If the target BRN node is unknown, the receiving node triggers a route discovery mechanism to find the target. Then it routes the Ethernet packet using the known route. The employed technique is called bridging.

Since all this happens at Ethernet level, BRN nodes do not need to have IP addresses assigned. In fact, for various reasons the BRN is desired to work without any centralized resources and even a \rightarrow DHCP server would be such a resource.

Hardware

Because the BRN project aims to cover a vast area with its wireless mesh network, it needs many inexpensive nodes. Thus normal commercial hardware routers with integrated access-points are used: Linksys WRT54GS and Netgear WGT634U. Both have a built-in WLAN interface and a 5-port Ethernet interface.

Software

A modified OpenWRT is used as distribution for the base system. It is an open source distribution for the Linksys WRT54G platform. It was ported to the Netgear WGT634U platform by BRN project participants.

On top of OpenWRT runs the click modular router software. It allows to capture, process and produce packets using reusable filter modules. The click software is explained in chapter 5.1 on page 37

Chapter 2

Problem Statement

This thesis is about the design of a Software Distribution Platform (SDP) for mesh networks in general and includes an implementation for $\rightarrow BRN$ in particular. SDP tries to solve the problem of reliably and securely distributing software within BRN, taking into account its nature of being a decentralized ad-hoc mesh network. The software versions shall remain consistent across the BRN as long as possible. This includes solving the following sub-tasks

- notifying nodes about software updates
- transferring files
- verifying package integrity and authenticity
- synchronizing system clocks
- switching software versions

While security is generally included in the discussion, it is not the primary concern. Also the OpenWRT/OpenWGT system hardware and software is only covered to better show, how SDP integrates into it.

An Example

To better illustrate the problem, here is an example of a situation in that SDP might be used.

In an initial stage, 20 nodes might be setup with initial software that employs the Dynamic Source Routing (DSR) protocol and 15 of them are deployed by different people. Most of them are installed on roofs for operation and another

3 nodes are given away to people who did not activate them, yet. Later on it is found that an additional option to DSR or a completely new protocol will perform better, thus an update package is prepared by BRN project members.

Now the problem is to distribute the new software to all existing nodes without leaving the BRN in a state with mixed, incompatible versions for a noticeable time.

Chapter 3

Current State of the Art

Of course, software has been distributed before by a wide range of methods. This chapter will start out by giving a summary of today's commonly employed methods, describing the common principles of the whole software update process and finally provide more detail about a few of those existing technologies to improve the reader's understanding about differences and similarities.

Note that SDP is only included in this table to give an easy overview on how it relates to existing techniques. The next chapter will then discuss which of the current methods can be employed to meet SDP's requirements.

Figure 3.1: Overview of properties of software update methods

| system | source | binary | delta | interact. | auto. | pull ¹ | push ¹ | decentr. |
|---------|--------|----------------|----------------|-----------|-------|-------------------|-------------------|----------------|
| MS Win | - | x | - | x | x | x | - | - ³ |
| SUSE | x | x | x | x | x | x | - | - ³ |
| Gentoo | x | - | - | x | x | x | - | - ³ |
| FreeBSD | x | x ² | x ² | x | x | x | - | - ³ |
| Netgear | - | x | - | x | - | x | - | - ³ |
| Netboot | - | x | - | - | x | x | x | - ³ |
| P2P | x | x | x | x | - | x | x | x |
| Worm | - | x | - | - | x | - | x | x |
| SDP | - | x | - | - | x | x | x | x |

¹push and pull denote which side initiates a data transfer. Push means that information is pushed towards those that want it, for instance, sending email. Similarly a user requesting a page on WWW does pull.

²not by itself, but with binup [Perc2003]

³Updates can only be decentralized in so far as a local mirror can be setup anywhere — usually in large administrative entities — and afterwards be run independently from the vendor.

Software update distribution methods can be categorized into:

- offline - e.g. distribute CDs to people (as in use in some current mesh networks)
- download - let owners download first and initiate updates later (as for Netgear router firmware)
- online - update automatically (the only viable variant for BRN as will be shown later)

Additionally, software updates are distributed in one or more of a different formats:

- as source code
- as binary package
- as binary delta or difference package

Previous large scale software distribution methods include:

- Microsoft Windows updates [WhIr2004],
- Gentoo Linux updates from source code,
- UNIX or Linux updates (e.g. SUSE or FreeBSD [Perc2003]),
- Booting from network over NFS or using software from SMB or OpenAFS shares
- Automated Software Deployment in a Large Commercial Network by Digital, see [Tall1995],
- Embedded device updates (e.g. original Netgear access-points, Nokia, ...).

Those methods follow common principles and steps:

1. Software is packaged into a common well defined format and tested prior to delivery. Examples for package formats are **rpm** for SUSE Linux and **msi** for Microsoft Windows.
2. Often notification about newly released updates is sent by email.

3. Software → *packages* are transferred to clients — offline (DVD/CD/floppy) or online — either automatically or user-initiated, as with Netgear updates.
4. Clients try to install software and might fall back to the previous version if installation fails. Installing often includes executing the pre-install and post-install scripts embedded in the package.
5. Clients stop old software and start the new one. Sometimes a reboot is necessary to complete installation.
6. Clients uninstall old software, which often includes executing the pre-uninstall and post-uninstall scripts from the old package.

P2P

Apart from the aforementioned there are also peer-to-peer (P2P) techniques like Avalanche and BitTorrent [GhRo2005, Cohen2003] can be used to distribute software. They are decentralized, because they can work without a central server. Some P2P protocols even combine push and pull mechanisms to enhance performance.

P2P improves overall network performance and robustness by letting user's machines — named peers — connect to other peers that already have parts of the wanted data. P2P distribution is decentralized and also utilizes hashes to verify integrity of the transferred data. However most real world P2P applications rely on IP and routing.

Worms

Worms (sometimes mistakenly named “viruses”) also distribute software — that is themselves. They usually distribute without user interaction and often spread very rapidly throughout Internet hosts. Infection based distribution seems to work well.

The difference to regular software updates is of course that computer worms are malicious software that enters the system through unintended channels and often causes damage to the user by spying on him, opening backdoors for remote-controlling the machine or deleting his files.

3.1 Standards

There are well established standards for most of the required tasks, but it will be seen that not all are usable in our situation.

Asynchronous Notification can be done with a broadcast packet, or a normal TCP/IP socket.

Also for pure file transfers there are several established standards: HTTP, FTP, and finally the trivial file transfer protocol, TFTP [HTTP, FTP, TFTP]. Some update mechanisms also use the rsync or CVS protocols [Trid1999, CVS] to efficiently keep binary or source file collections up-to-date.

For integrity and authenticity verification there exist several well-proven cryptographic hashes (MD5, SHA1, Tiger, ...) and cryptographic signing functions (RSA [RSA1978], DSA)

Time-synchronization can be done with the Time Protocol [TimeProt] and continuous high-precision time-synchronization with [NTP].

For packaging installable software there is a variety of different software package formats in heavy use today and all of them employ compression to save storage space and transfer bandwidth. There are: deb (Debian GNU/Linux), rpm (Redhat/SUSE/Mandrake Linux), msi (MS Windows), zip, tar.gz (generic file collection)

3.2 SUSE Linux Update

To give a more detailed view on common software updating procedures this section will describe SUSE Linux updates. It is likely that other commercial software vendors use very similar methods.

The package format

SUSE Linux distribution employs the Redhat Package Manager — rpm format. First here is an overview of rpm features, as most other software packaging formats have similar characteristics.

rpm packages contain various useful information about their content:

- digital signatures for the content

- list of contained files
- list of required packages and files
- pre-install, post-install, pre-uninstall, and post-uninstall scripts
- various additional information like uncompressed file size, packager Name, build date, ...

Here is an example to illustrate the format: **rpm -qi rpm**

```

Name       : rpm                               Relocations: (not relocatable)
Version    : 4.1.1                             Vendor: SUSE LINUX Products GmbH
Release    : 208.2                             Build Date: Sa 11 Jun 2005 01:53:04 CEST
Install date: Di 18 Okt 2005 13:29:37 CEST      Build Host: purcell.suse.de
Group      : System/Packages                   Source RPM: rpm-4.1.1-208.2.src.rpm
Size       : 5970541                           License: GPL
Signature  : DSA/SHA1, Sa 11 Jun 2005 01:58:41 CEST, Key ID a84edae89c800aca
Packager   : http://www.suse.de/feedback
Summary    : The RPM Package Manager
Description:
RPM Package Manager is the main tool for managing the software packages
of the SuSE Linux distribution.
```

RPM can be used to install and remove software packages. With rpm, it is easy to update packages. RPM keeps track of all these manipulations in a central database. This way it is possible to get an overview of all installed packages. RPM also supports database queries.

To disambiguate packages with the same software for different platforms, file names of rpm archives contain the architecture the package was built for. Examples are `xorg-x11-devel_6.8.2-30_i586.rpm` — “i586” for a binary package for Intel Pentium and compatible and another example being `pwdutils-3.0.4-4.2.ppc.rpm` for Power PC.

This file was built from a platform independent source package named `pwdutils-3.0.4-4.2.src.rpm`.

The rpm database keeps track of all installed rpm packages, their versions, dependencies, files and their MD5 hash values. Thus it is able to warn, if an incompatible package would overwrite existing files.

Update policy

Most updates are supplied to solve security problems, but sometimes updates for other flaws are provided, if they render the software or system useless. This discussion will focus on the security update policy, but most of it also applies to normal updates.

SUSE, now owned by Novell, operates a security team who watch various sources about newly known vulnerabilities. They classify the threat potential based on the risk assessment:

- Is the vulnerable software installed by default or commonly manually installed like apache?
- Is the vulnerability exploitable by remote attackers?
- Is it easy to exploit or does it require special knowledge or tools?
- What would be the impact? Root access, foreign code execution, data manipulation, information leakage?

The examples of worst cases so far were

- a remote exploitable problem with sshd which is installed and started by default and allowed a knowledgeable attacker to become root.
- a bug in the crontab program, installed by default on SuSE 7.1 that was trivial to exploit and gave root access
- a bug in the chfn program, installed by default on SuSE 8.2, 9.0, 9.1, 9.2, 9.3, 10.0 that was trivial to exploit and allowed a local user to become root.

After classification, updated packages are built — first for most urgent problems. Packages are built by applying a security patch to the old software versions that were shipped with the respective release of SUSE Linux. Usually there are about 8 different SUSE releases to be maintained at any time. Only in rare cases when the problem can not be solved with a simple patch a completely new version of the package is built. This exception applies to closed source software like the Adobe Acrobat Reader, but also to non-trivial flaws in complex systems like the mozilla web browser.

The package name contains a build counter which is automatically incremented for every newly built version, so that the packages are automatically distinguishable.

Updated packages are not released at that moment, but tested internally at SUSE for a few days to guarantee a certain level of functionality and stability on all of the supported platforms. Due to this quality assurance, only few cases occurred, when security updates introduced new problems.

Once the update package is deemed stable the security team uploads the .src.rpm package and all pre-built binary packages for all supported

architectures to the update section of the respective release version on `ftp://ftp.suse.com` and afterwards sends an announcement to the `suse-security-announce` mailing list. This announcement includes MD5 hash values for every new package as well as a PGP signature to provide verifiable authenticity.

The update process

The main update utility for SUSE Linux is called Yast Online Update — `YOU`. In the default installation it has to be called interactively, but can easily be configured to run automatically and unattended every day or every week.

On start, `YOU` will download a current list of URLs of mirrors and randomly choose one of those, to distribute load amongst all mirrors. Either FTP or HTTP will be used for all following file transfers, depending on this mirror URL selection.

Next `YOU` will download new update descriptions from the selected mirror and automatically determine which of them are applicable using the list of installed packages as reference.

During interactive operation the user can then disable updates he wants to keep back or enable additional updates. Automatic updates can skip updates with pre-installation information or just use the complete list without changes.

In the next step `YOU` downloads updates. Recent versions of `YOU` first check, if the currently installed software version is unmodified and then only request the binary delta rpm package from the mirror server. As fallback, full update packages with both updated and original files are downloaded.

For every delta-rpm the full update package is reconstructed locally by

```
applydeltarpm new.delta.rpm new.rpm
```

For every downloaded file the embedded signature is checked using the pre-installed public `suse-build-key`: **`rpm --checksig new.rpm`**. This guarantees authenticity and integrity of all new software.

Afterwards the new packages are installed as updates with **`rpm -U new.rpm`** taking advantage of the regular package manager facilities for updating single packages from local sources.

Finally, after all new packages are installed, `SuSEconfig` will be called to do some SUSE-specific cleanup and config file processing.

3.3 Other Methods

The following section tries to give an overview of other commonly used methods for installing and/or updating software.

3.3.1 Package Formats

msi

Microsoft developed the Windows Installer and its msi package format since about 1995 using a structured file format.

msi files employ a relational database. The goal of msi is to provide a consistent installation mechanism and interface to both users and developers. Also it allows uninstallation, repair and is able to revert changes, if an error occurred or the installation was canceled by the user. This is possible with the transactions provided by the database.

Microsoft Windows 95, 98 and ME did not ship with the Windows Installer program `msiexec.exe`. Thus its users needed to download the Windows Installer itself once, before those versions of Microsoft Windows were able to use msi software packages.

By itself, the msi format does not provide for determining or downloading dependent packages or verifying authenticity of the package's content, but it is thinkable that other utilities extend this file format. Also it seems common to include all necessary libraries in the msi file so that users will not need to download or install other components separately.

emerge and ebuild

Gentoo Linux uses small "ebuild" files to hold information about a single package. The distribution has thousands of such ebuild files that comprise the "Portage Tree" — Gentoo's archive of readily available software.

Every ebuild file includes a summary about the content, dependencies on other packages and information that allow to automatically build and install the software from source code. Sometimes Gentoo-specific patches are also provided to be applied before a build.

Gentoo Linux uses the `emerge` utility to easily manage all package-related operations.

Note that this is very different from most other commonly used software distri-

bution methods in that platform independent source code is provided and only compiled for the target platform with user-defined options on demand. It also allows users to customize packages with various compile time options. Gentoo does that by means of the USE variable.

This and much more is covered in the documentation that can be found online: <http://www.gentoo.org/doc/en/handbook/>

Updating Gentoo Linux

Gentoo Linux is one of the few systems that almost exclusively rely on the Internet or local area network for distributing updates, while all other system mentioned in this chapter (not counting P2P and worms) can easily be updated using downloaded package files from a CD.

Updating Gentoo is normally based on updating source code. The process is thus:

1. **emerge --sync** this uses rsync [Trid1999] to efficiently update all ebuild files.
2. **emerge --update --deep world** this downloads the associated files over HTTP or FTP, builds and installs all updated packages from the new sources.

For both steps configurable mirrors are used. Selecting a mirror from the own region is simplified by providing DNS entries like `rsync.europe.gentoo.org`. This particular one resolves to mirrors in Europe.

The emerge utility also supports building and installing binary packages, so that administrators with more than one Gentoo PC can save time by compiling only once and installing the resulting binary file on all machines. However, the distribution itself does not provide pre-built binary packages⁴.

Gentoo uses MD5 hash values to verify integrity of source files and patches, but does not provide for authenticity of files, yet. However, authenticity verification with gpg is currently in development.

⁴except for the Gentoo Reference Platform, but these packages are only updated with a new release after months

3.3.2 Embedded Device Firmware Upgrades

After all this detail on PC operating system upgrades, this part will give a short overview of known firmware package formats for embedded devices. Since BRN's designated target hardware is also such an embedded device, this bears significant relevance for this work.

These methods have in common that the user initiates a download of a firmware image from the vendor's web site using an ordinary web-browser. This file is then manually uploaded to the embedded devices.

Embedded devices are pretty different from PCs. Unlike PC software that may run on a wide range of different heterogenous hardware, embedded systems have more or less homogenous hardware and the hardware manufacturer also provides the official software and updates. Additionally embedded devices are usually designed and used for a more special purpose.

AVM Fritz!Box 7050

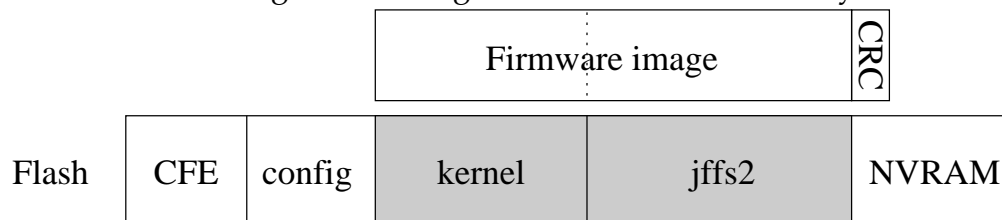
The AVM Fritz Box update packages are `tar` archive files that are uploaded through HTTP to the device, after password authentication. Once the file is completely transferred, the archive is unpacked into RAM and the `install` script is executed.

Netgear WGT634U

[Netgear] firmware packages contain a complete image of the new system. This consists of a Linux kernel binary with a compressed minix `initrd` and a `jffs2` (Journaling Flash File System, version 2) file system image. After authenticating at the web interface using a previously set password, the image can be uploaded through HTTP. The update mechanism verifies integrity using an embedded CRC32 checksum. Afterwards the kernel and `jffs2` areas on the embedded flash are erased and then overwritten. Flash and update file layout is illustrated in figure 3.2.

CFE is the equivalent of a BIOS (Basic Input Output System) for the device. The config area stores all persistent settings of the running Linux system like IP address and wireless network name and finally, there is the NVRAM that stores all settings for the CFE like MAC address, kernel location or bootloader options.

Figure 3.2: Netgear Flash and Firmware layout



Linksys WRT54GS

Updates for this device are very similar to the Netgear system, in that a complete image is transferred and flashed. For this system a compressed cramfs filesystem is used that can be efficiently decompressed on demand by the Linux kernel.

3.3.3 Current Mesh Network Updates

Since there are a few working mesh networks operating in the world and even in Berlin it would be interesting to know how they distribute and install new software.

freifunk.net

Like some others freifunk.net is employing OLSR to build a wireless mesh network. Firmware can be freely downloaded by any user from <http://www.freifunk.net/wiki/FreifunkFirmware> and be manually installed. Sometimes this non-trivial installation is helped by others in an installation-party — this was also done by other related projects like www.paris-sansfil.fr. Distributing software updates via CD is also thinkable, but details could not be found.

wlanhain.de

wlanhain.de — though similar in spirit to other mesh networks like BRN — is no ad-hoc mesh, but a simple pre-planned collection of free access-points using ordinary client-server connections. For this reason they do not need any special software at all, but just the →WLAN drivers for the 802.11b compatible hardware.

Chapter 4

The Software Distribution Platform

This chapter deals with the Software Distribution Platform — SDP itself. At first, the requirements are analyzed, then the applicability of current methods is discussed and finally the different SDP specific components are introduced and explained.

4.1 Requirements

In chapter 1.6.1 on page 6 you could read that the BRN needs a well designed SDP to be useful as a testbed for distributed protocols and what the constraints on the BRN are. From these most requirements for SDP arise.

The SDP must provide reliable and cross-compatible updates, which means updating must still work with a broken routing protocol or different (thus incompatible) routing protocols around the network. This would be the case after a temporal network split, which might happen for many exotic reasons including, but not limited to a local power outage, a damaged aerial, and/or rain absorbing 2.4GHz radio signals. The reliability and robustness requirement is covered in chapter 4.2.2 on page 30.

Additionally SDP has to be cross-compatible — beginning with the first released version (let us call it “version 0” ; see chapter 4.2.5 on page 31 on versioning) to allow upgrading a very old node to the most recent version. If this were not the case, updating old/broken nodes would pose an enormous problem¹.

¹One would first need to locate those nodes and then they might be installed on roofs or other locations difficult to reach. And even when a node is in your physical reach, updating such a broken node, might still pose a problem, because they usually do not have a known IP address assigned to any interface, no VGA - only maybe a serial console.

Another crucial requirement is that the SDP must not allow malicious nodes to distribute and install unwanted software. This is covered later in chapter 4.2.7 on page 33. Also it is desired to be able to limit the changes that SDP can do to a node.

Besides those security considerations, the SDP should also try its best to make sure that all nodes of BRN's distributed mesh run the same version of software most of the time – ideally every second. This is covered in chapter 4.2.6 on page 32. Otherwise incompatible routing protocols would cause a degradation in connectivity between the nodes, possibly splitting one part of the network from another until software versions are updated on a sufficiently large part of the BRN.

So summarizing the requirements for later reference, SDP must:

1. be cross-compatible,
2. be robust,
3. maintain consistent versions,
4. be secure.

4.2 SDP Design

Designing SDP (as with any sufficiently complex software component) involved a number of design decisions, choosing one method from several ideas, in order to best meet the aforementioned identified requirements.

4.2.1 Using current methods

How can established methods be used for SDP?

SDP in fact utilizes many of the common practices of updating software listed in chapter 3 on page 13, but modifies and extends them when necessary to fit the requirements specific to BRN.

Especially the methods for notification and file transfer — step 2 and 3 from the common principles for updating, found in chapter 3. Using a centralized storage is not possible for the decentralized BRN. Furthermore, most existing transfer methods can not be used for SDP because they rely on IP and routing,

which makes them unusable or at least less robust to use in BRN, because BRN experiments with new routing methods - and they might sometimes fail.

Also SDP can not rely on users to download and install new software packages. While this is fine for a single user and his home network, it is unsuitable for BRN because it would need days or weeks until the network reaches halfway consistent versions.

If you attentively studied the comparative table given in chapter 3 on page 13, you will have realized, that SDP is most similar to a computer worm distributing itself. Just like it, SDP employs an infection based distribution method, which is simple, decentralized, robust and effective. Most alternative approaches like contacting centralized servers or selected “super-nodes” like some P2P protocols do, are not applicable to BRN because we can not rely on routing.

This infection-based method involves injecting a new software version at any node of the BRN, which then gets distributed to its neighbors and from there to other neighbors until the whole (sub-)net has the new software version available, but not activated at this point. This delayed activation will be explained later in chapter 4.2.6 on page 32.

The advantage of this infection based mechanism is that each node only needs to communicate with its direct neighbors over a direct WLAN/Ethernet link, thus no routing is involved, which even allows old nodes with incompatible routing protocols to be updated. It uses BRN packets in raw IEEE 802.x frames so that no allocation of unique Internet Protocol addresses are needed. It also does not need a centralized data storage (file server), which might otherwise fail when the mesh grows to hundreds or thousands of nodes, all requesting the new version at a time. Additionally it saves transfer bandwidth by only transferring a new file once over a link, leaving more bandwidth to BRN and the applications running within it.

You may notice that the situation differs from a local area network of private PCs, that usually have heterogenous system hardware, partition layout and varying installed software. It is also different from PC installations in enterprise environments that may have homogenous hardware and software, but usually have only one administrative authority (e.g. the IT department). Also in case of failures, PCs are usually easier to reach and repair. Even in such static networks distributing software to maintain consistent software versions everywhere is still hard as can be read in [Tall1995]. As a result some of the PC-based ideas can not be applied to our problem.

How to distribute

There are some alternatives to distribute software

- offline - e.g. distribute CDs to people
- download - let owners initiate updates
- online - update automatically over the BRN

Of course the offline method (that is in use nowadays) would need days, if not months to spread a new version and would therefore result in different software versions and thus fail to meet the criterion of version-consistency, which is requirement 3 for SDP. That makes it absolutely unsuitable as testbed.

The offline method shares some problems with the updates per download variant. Node owners can be away on vacation, be unreachable by e-mail or just too busy to spend time on updating a BRN node. Again, this would result in long periods of mixed versions that possibly break the network routing.

So the only viable variant for updating a whole network would be to automatically install software on the nodes.

What to package

There are in principle three different ways of updating.

- Major updates,
- incremental updates and
- a difference to version 0.

A difference update referring to version 0 (which means the software initially installed on the distributed hardware) is smallest and easiest to create. It would only include all files which differ from files in version 0 and be independent from previous updates. Difference updates require some special system layout on the nodes to allow removing the previous difference, thus restoring version 0 and thereafter updating to the new version. Software that would allow layering directories is available, but not completely stable: translucency Linux kernel module (written by myself) and the unionfs²

²union filesystem home page: <http://www.fsl.cs.sunysb.edu/project-unionfs.html>

An incremental update would be like a difference update, but also include all files, which were changed in previous updates. This would ensure that systems with version 0 can update to any version i first and then to a later version j by just overwriting all previous files with new ones. The advantage of this method is its simplicity. It does not require any special mechanisms on the target node to keep version 0 separate from the updated files. The disadvantage is that creating such packages is slightly more difficult and the package volume might be increased compared to difference updates.

Major updates would be extremely rare (ideally never). Such update would include downloading a full system image from a neighbor, verifying its hash and signature, writing it to persistent storage (flashing) and finally triggering a reboot to activate it. Great care is required when coding such an update. Additional testing in a small mesh is strongly recommended. In case of failure the systems can still be booted over TFTP on the wired link, but this is no viable option for a full-scale deployed network like the one BRN aims to become.

The current implementation of SDP on the nodes is flexible enough to allow for all 3 updating variants. The only missing thing is code to transfer and update a complete image for major updates.

For simplicity of version management, software versions are distinguished by a single positive integer number that is usually increased by 1 for each public release. As long as no larger steps are made, the version-space of 32 bit will suffice for 136 years, even if one new version was released every second. That is why wrap arounds of version IDs need not to be considered. As another simplification, downgrading software needs to be done by upgrading to the next version. Only the meta-info file needs to be transferred for this to work. This way the SDP version management is simplified to fetching versions with a higher ID and switching to them at the right time. In the first working SDP version, the switch was done immediately to allow other components like time synchronization and asynchronous software switching to be implemented later.

Another point was a combined push/pull notification mechanism. A push notification is useful after a new version has been received, to immediately tell neighbors about it and allow it to spread as fast as possible. This gives total distribution time $n \cdot T$, with T being the time needed to transfer software from one node to another and n being the longest distance (number of hops / path length) between the injection point and any other node in the network. Of course in practice two concurrent transfers interfere with each other, so that the total time is a bit longer.

A pull - that is a node requesting another node to send its version - is needed for requirement 3 to let nodes check for new software after a power-on to avoid running a possibly incompatible version longer than necessary. Additionally regular broadcasts (push) of currently available and running software versions make sure that the whole network converges to one version after some time, even when notifications are lost during transfers, thus contributing to robustness.

4.2.2 Robustness

This section deals with requirement 2 from chapter 4.1 on page 25.

BRN nodes may be physically hard to reach and might sometimes have faulty routing software installed. Also they do not necessarily have an unique IP address assigned to their wireless (802.11g) network interface which prevents remote logins with normal software. From what has been said about BRN requirements it should now be clear that the SDP must be very robust. Failing would mean additional work and frustration for people and thus possibly causing them to stop supporting BRN.

To deliver software updates reliably under these circumstances, SDP must not rely on IP and routing to work. This is solved by designing SDP to utilize an infection based distribution method, which allows it to work completely without IP and routing by exclusively communicating with direct neighbors and transmitting everything in BRN packets in Ethernet (IEEE 802.3) frames.

Another aspect of robustness is reliable operation. If BRN nodes crash for any reasons, it might be hard to bring them back online by resetting. To improve node reliability, we made a watchdog script that would reboot the node after 2 seconds of non-responsiveness. While this would handle most cases of a kernel crash, or memory allocation problems, it can not easily and reliably detect network connectivity problems. Also such connectivity problems could be caused by certain circumstances or environmental influences like radio interferences, a defective neighbor node or a broken antenna. Additionally neither of these problems can be solved by software.

In an attempt to improve robustness a fallback mechanism was added to SDP. If a node does not receive SDP beacons for some time, it will call a script (`/var/update/fallback_stage1`) which then can try to start alternative software or would enable a telnet daemon, so that remote administration is possible over the wireless interface. This allows for both diagnostics and repair op-

erations without needing physical access (e.g. climbing on roofs) and thus can be a great help to promote BRN.

While this sounds great at first, it also has its pitfalls. When the mesh is weakly connected, it can cause nodes to go into fallback mode when they do not see neighbors for some time (e.g. because of temporary power outage). This in turn reduces the number of neighbors of other nodes which then may go into fallback mode themselves because they no more receive SDP beacons.

So from what was said in the last paragraph it should be clear that the fallback mode should still have SDP enabled (thus click software running) so that it can still automatically receive better software and/or switch back to normal operation when appropriate.

It might even be possible to integrate a telnet server into the fallback click software or let it have some IP address assigned that is derived from the unique MAC address. None of these more advanced fallback methods have yet been implemented for lack of time.

4.2.3 Notification

As mentioned earlier, SDP has to work without IP. Thus SDP uses local Ethernet or \rightarrow WLAN \rightarrow broadcast packets to notify neighbors about the current version identifier. Including other information is not necessary, because it is included in the signed — and thus trusted — `meta-info` file that can be requested using the version ID and the source MAC address. Taking untrusted information into account could pose a problem for security.

4.2.4 File Transfer

For file transfers a variant of TFTP was implemented, based on the official TFTP³ RFC 1350 [TFTP] but using BRN packets instead of UDP/IP datagrams. See chapter 5.4.1 on page 46 for details.

4.2.5 Versioning

The aim of SDP is to distribute, install and run new software on all BRN nodes. To determine, if a software package is really new or to find out what nodes run

³the Trivial File Transfer Protocol is datagram based (UDP instead of TCP) and implements simple error detection and recovery

what software, there is a software version identifier included in every software package. This is also important to tell apart different (possibly incompatible) versions of the same file.

The version identifier is a simple integer number that is incremented with each new release. This brings up the question: Why not use more complicated versioning like with other software - e.g. 1.23-4e?

This naming scheme is used to signal to **humans** how stable the software is and how major the changes between two versions are. This is no advantage for this application. Node software needs to determine automatically and without user-interaction if a certain software is newer than the current one and this is easily accomplished by comparing two integers. For instance version 100 is clearly newer than 98 and newer than 1, but this relation would be more difficult to establish for versions 1.1-5x and 1.23-4e. There have been reports of such problems with other software update mechanisms: <http://apt4rpm.sourceforge.net/faq.html#q33.1>

In fact, one can even define some semantics into the integer numbers, e.g. the last two digits could be a minor version number, but this is just a matter of interpretation and the updating mechanisms would not need to know about it. It does not care as long as the version identifier numbers are ascending.

The first publicly released BRN+SDP software version shall have a version ID of 0 just to give it a name that signals that there is no preceding version.

4.2.6 Consistency of versions

The problem of ensuring that all nodes run the same software version can be solved by first distributing a software package and later let all nodes synchronously switch to it. This works by embedding the activation time within the package and by synchronizing node system clocks with each other and with UTC. Since BRN nodes do not have unique IP addresses, this could not be done by some standardized protocol like NTP. So a simple custom BRN-NTP protocol was employed for time synchronization using broadcasts. See chapter 5.2 on page 40 for discussion of the implementation.

Additionally it is possible to distribute and run experimental software versions, which have a duration value embedded in the package. A typical value would be 3600 for one hour of testing. After that period, the previous version would be switched back on all nodes at the same time. See chapter 5.3.1 on page 44.

4.2.7 Security

Security is manifold. To ensure having a secure overall system, one often needs to provide integrity, authenticity, and confidentiality. Because the BRN software is open source and can be downloaded anyway, transfers do not need to be confidential, but the other requirements still have to be met to provide a secure system. The essence of the following is this: We do not want anyone to tamper with our software.

Signing for authenticity

There might be malicious users and nodes that could inject their software to spy out other people or even take over the whole BRN. To prevent this, each software package must be digitally signed by the CA key (or by a developer, whose certificate is sent along with the package and is signed by the CA). The CA's public key is pre-installed on every SDP node and is considered as ultimately trusted. After reception of a new meta-info file the node will first verify authenticity of the update and discard it otherwise. However, with this alone it is not possible to give each node a verifiable identity. This would require each node to have its own secret certificate, which is signed by the trusted CA. Verifiable identities might still be implemented with some registration process for independent node operators but this is outside the scope of this work.

The OpenSSL package already has all necessary code and is available for the OpenWRT platform. SDP signs the SHA1⁴ value of the meta-info file (which contains SHA1 values of all other files). The RSA⁵ public key encryption algorithm is used for signing.

The only downside to using OpenSSL is its relatively large size. It is possible to use a stripped down version of OpenSSL with only the relevant algorithms (RSA, SHA1) included. Alternatively there are tiny SSL libraries like libmatrixssl, but as it seems it is not interface compatible and would need some development effort to integrate.

A sidenote on the employed SHA1 and RSA algorithms: both have been in common widespread use for a long time and even though some successful attacks have been published in the past, they can still be considered sufficiently secure

⁴see [SHA1] - a commonly used secure hash algorithm standardized by NIST - if the data changes, the hash value changes as well (unless the hash function is bad)

⁵see [RSA1978] - Rivest, Shamir, and Adleman published this public key cryptographic algorithm in 1977

for BRN.

Indirect signing

Indirect signing is the more flexible variant of ensuring authenticity of software update packages. In this scenario the BRN's root-CA only signs developer certificates and those are used to sign software packages.

For this to work, the developer's certificate must be appended to the meta-info file. It is first verified using the root-CA's public key and afterwards used to verify the hash of the meta-info data.

This adds another element of complexity: as developers may join or leave the team, it must be possible to revoke their certificates individually. Certificate revocation lists (CRLs) are commonly used for this, but as there is no centralized data store in BRN the CRL needs to be regularly updated within software updates. Upon changes in the CRL, it is advised to release a new software package with only the CRL file changed in it, so that afterwards the former developer will no more be able to deploy software on BRN nodes because his certificate can be found on the CRL and is thus regarded invalid.

Due to the complexity of this non-essential feature it is not implemented in my SDP code.

There is another, much better way to implement indirect signing. This works by having the CA automatically sign software packages, if they are signed by the appropriate developer's or sub-CA's certificates. In that case it is no problem to keep the CA host's CRL updated and consistent. With this setup is even possible to implement more complex setups. E.g. software distribution policy can demand a signed test protocol for a new version to be accepted and the software on the CA machine can adjust version numbers to not collide with those of other developers.

Summarizing this section, it is possible to verify software authenticity on BRN nodes using OpenSSL and even employ flexible indirect signing.

Avoiding DoS

In addition to employing signatures, great effort has been put into not allowing erroneous software or malicious users to damage the flash memory by repeatedly writing data to it (flash memory allows for approx 100000 write cycles). This would not only allow for a remote denial of service (DoS), but permanently damage hardware. This is achieved by storing received files in a dedicated tem-

porary directory on the RAM disk and only copying it to flash after successful verification of authenticity.

The aforementioned provides for security, but there is also demand for safety. E.g. one would not want an update to change arbitrary files or corrupt the filesystem. This is nearly impossible to implement on the nodes without dramatically decreasing flexibility. With the activation script being executed on the node it is (in theory) possible to make any change to the system. But for the activation script to be valid, it must be accompanied by a meta-info file, that must be signed by the CA or a trusted developer. This way it is possible to enforce a certain policy before releasing any software to the public. Such policy could require some minimum testing period on a small test mesh, let humans review code changes, the activation code and other aspects that can not be decided on a BRN node.

Atomicity

Another important aspect was to make write accesses to persistent storage (flash) as atomic as possible. Atomicity is a well-known term in database systems, but usually is not fully supported by most operating systems and their filesystems. As an example of non-atomicity imagine, we would install new software by copying several files (totaling one or two megabytes) to their final location and in the middle of a file, the power would be switched off. Writing so many data takes some time and is usually done in blocks of a certain size - e.g. four or 8 kilobytes. Now, when the machine boots after the power-failure, the jffs2 will have ensured that the filesystem is still intact, but due to non-atomicity of the writing operation the routing software would be incomplete. In contrast, renaming a file or changing a link is near atomic, as it is one syscall, which only needs to change a few bytes in one location which is possible in milliseconds rather than seconds. For this reason, placing new software into the persistent software library is done in two phases: copying to a sub directory in a temporary directory on the flash, which is emptied at reboot to avoid partial files cluttering up the filesystem renaming (moving) the sub directory to its final location

Chapter 5

SDP Implementation

Figure 5.1 gives an overview of the SDP components and their interaction.

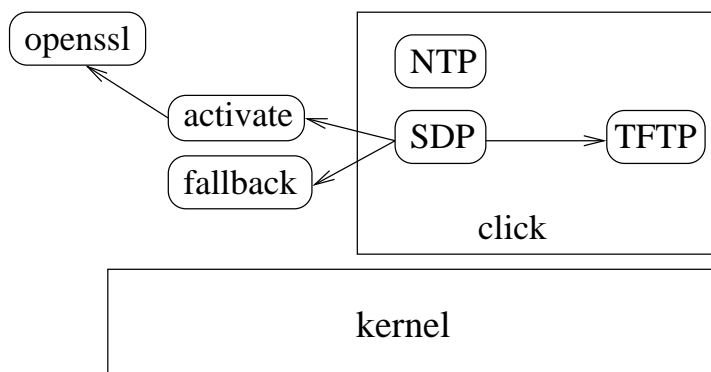


Figure 5.1: Overview of the SDP components and their interaction

The complete SDP source code is provided for reference and for interested readers at <http://bmwthesis.webhop.net> and <http://www.zq1.de/bmwthesis>. It is licensed under the Gnu General Public License version 2 which explicitly allows use, modification and redistribution of the code as long as you open the sources.

5.1 Click Modular Router

Since both BRN and SDP build upon the click modular router software this section will give an overview of it before continuing with the actual SDP implementation part.

The basic click software was designed, implemented and documented by MIT.

See [ClickDoc]. Click models a router as a directed graph of filter elements through which the packets flow.

It ships with a comprehensive repository of readily available modules, named “elements”, while still allowing to extend it with customized modules written in the C++ programming language. This makes it powerful and versatile while still being comparatively easy to use.

Both, BRN and SDP are implemented as separate click modules which can be loaded at runtime to provide the project-specific element classes.

Once the modules and elements are ready, click easily allows to arrange them in filtering graphs by just using them in a click configuration file. Figure 5.2 is a visualization of such a graph, created with the `click-viz` and `graphviz` tools. It includes the **BrnSDPGEN** element to generate SDP packets with BRN-NTP time synchronization information. This is then transferred to the **DeviceClassifier** element which passes the packet to one or all of the output channels, depending on the `udevice_anno` annotation field of the packet, with the source MAC address being set to the one of the respective output device. The **Queue** then stores the packet until the **ToDevice** element is able to send the data on the physical media. In click’s technical terms it is needed to connect a push device like the **DeviceClassifier** to a pull device like the **ToDevice**.

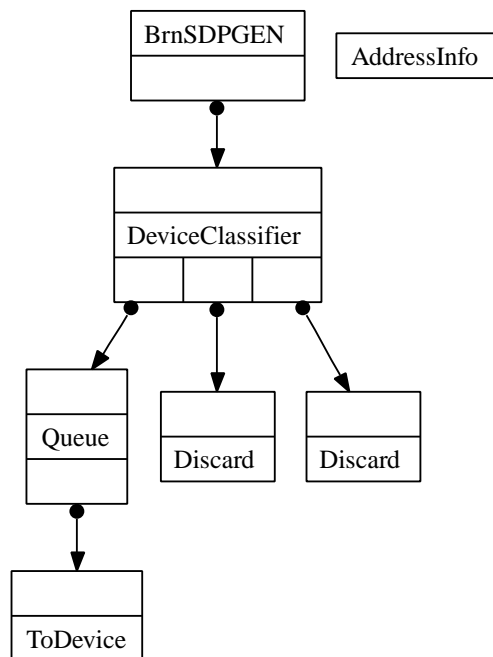


Figure 5.2: Click graph of a BRN-NTP time-server

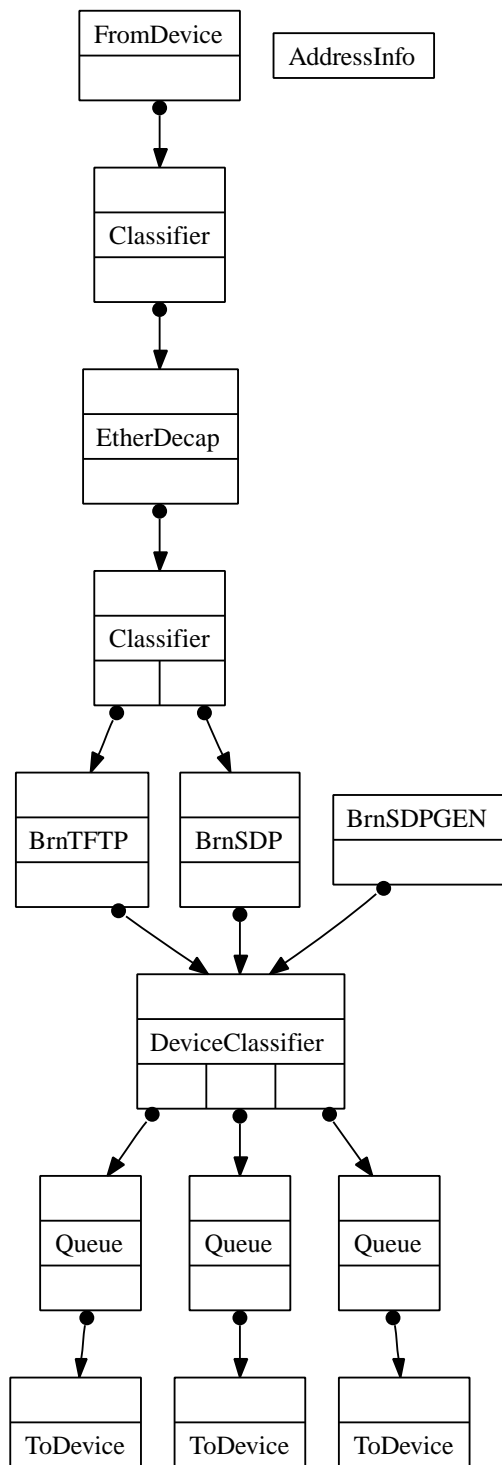


Figure 5.3: The SDP click graph

A more complicated click graph with the data flow of all SDP click elements is shown in figure 5.3 on the page before.

You find the corresponding configuration file in appendix B on page 57.

The BrnTFTP element includes both TFTP client and server handling code, because most of the header-parsing code is the same for both cases.

5.2 Time Synchronization

For first tests the distributed software was immediately activated to not require nodes to have synchronized system clocks.

This was not sufficient for the final SDP. Therefore the most simple stateless time synchronization protocol possible — named BRN-NTP — was implemented.

BRN-NTP works as follows: Each SDP beacon broadcast packet has a timestamp added as 64 bit value. It uses 32 bit for seconds and 32 bit for microseconds. While this is somewhat similar to standard Network Time Protocol (NTP - described in RFC 1119 [NTP]) it uses a different transport (BRN instead of UDP/IP port 123) and a different sub-second time representation because this needs no conversion from the `gettimeofday` and to the `settimeofday` time representation.

Whenever a node receives such time information, it calculates the average from it and its own, does some sanity checks to avoid setting time to impossible values, like before year 2000, by mistake and then switches its clock subsequently sending the new time. It does not switch its clock, if the time difference is below a certain threshold (200ms currently) to avoid unknown network delays heavily skewing time with every transmission and adjustment.

You can find the actual C code for reference in Appendix A.2.

One additional problem of NTP in conjunction with click arises from using `settimeofday`, because click runs single threaded and does not make use of the kernel's scheduling capabilities. Because of this click improperly schedules timers immediately after a `settimeofday` shifts time into future (e.g. after startup) and thus the scheduled time is suddenly in the past. This adversely affects the fallback timer. It is worked around by changing the fallback script (see chap-

ter 4.2.2 on page 30) to only take action after being called 3 times. A cleaner way would be to patch click to be notified of time changes and adjust scheduling timers appropriately (not implemented).

Additionally it is desired that all BRN nodes have coordinated universal time (UTC) and not just any synchronized time. To achieve this, some special BRN nodes with other means of UTC time synchronization (e.g. GPS, NTP) can be attached anywhere to the mesh network. As an example the script `/usr/sbin/ntp-client` is provided in appendix A which uses `rdate` to regularly receive time information. Such UTC nodes do not change their own time based on received SDP time information packets if they have a `/var/updatelink/disablebrnntp` file. This approach has the advantage of being both decentralized and still functional without Internet Protocol routing. Even when all UTC nodes are disconnected from the network all other nodes should converge to one time which should remain close to the previous synchronized times.

Optimizing

To improve BRN's NTP performance, two special time values (both in microseconds) have been introduced:

- **mingoodtime** - values below this are known to be in the past, thus if the own clock is below, it is immediately set to a time above **mingoodtime** instead of the normal averaging procedure.
- **minvalidtime** - values below this are illegal and always discarded. It is set to 2 days after the normal start clock value of BRN nodes (00:00 Jan 1 2000).

Mathematical Analysis

As with many algorithms, one would be interested in its properties. Do times converge?

The simple algorithm to be tested is, which is slightly different from the actual current implementation:

$$T_{\text{self,new}} := \left(T_{\text{self}} + \sum_{j=1}^n T_j \right) \div (n + 1)$$

It can be shown that even with only one real UTC time source and all BRN nodes having very bad system clocks, the simple time synchronization protocol will result in convergence towards some time close to real UTC time.

To show this, let us assume, we have 1 node with n neighboring nodes, one of them being a 100% correct and reliable source (not adjusting clock from other nodes) and all other clocks running with a constant drift d . This drift assumption is a worst-case estimation. In the more general case some clocks might be running with lower drifts like $d/2$ or $-d$ but that would help the overall clock averaging and synchronization.

Network delays are neglected.

Every u seconds new time information is sent.

here is an example:

$n = 5$ five neighbors

$u = 60$ one update each minute

$d = 0.1$ that is 10% clock drift: a minute would have $60s \cdot (d + 1) = 66$ seconds

Now the bad clocks drift constantly by $u \cdot d$ between two updates. On the other hand we receive time information from n other nodes, and still have our own. So each update shifts the node's clock by about $1/(n + 1)$ multiplied with the current UTC-offset o towards UTC (the UTC node has $o = 0$ and a node with $d = 0$ would have $o = \text{const}$).

This means that if $o/(n + 1) = u \cdot d$ then the two effects will nullify each other. This lets us calculate $o = u \cdot d \cdot (n + 1)$ For the above example values we find $o = 60 \cdot 0.1 \cdot 6 = 36$. So even with 10% drift (which is extremely bad) nodes would only be 36 seconds away from correct UTC time. This is still an acceptable worst case offset for software distribution. Measurements showed that the real clock drift is normally only two seconds per hour ($d = 0.0005$). Given the above worst case estimation this is no problem, as long as there is a constant input of correct timing information.

Note that theoretically the above point of equilibrium $o = u \cdot d \cdot (n + 1)$ could become infinitely large for infinite n . But for BRN n is the number of nodes in radio range and when that is about 10, the whole surrounding space will already be covered with wireless signals and no new nodes need to be added.

Another case is that a node is multiple hops away from a UTC node. It can be inductively shown that if a node being $m - 1$ hops away from a UTC node is within some offset $(m - 1) \cdot o$ from UTC, then the node m hops away will be within offset o from its neighbor and $m \cdot o$ from UTC.

Finally, for a good time protocol one has to check that in a network without a UTC time source, clocks do not skew due to summing up of latencies at each time information transfer. Typical wireless transfers need 3ms - far below the 200ms threshold to change the local clock, so this should be OK. This mode of operation is not favored for the BRN, because software updates contain an absolute time for their activation. However, occasionally there might be temporal network splits that could let it happen. The connected sub-networks will still have synchronized clocks with each other, but there is no guarantee to what absolute time they will have.

The plot in figure 5.4 shows a measurement of a typical WRT54GS system with a quartz having a constant 0.04% drift. After 2750 seconds it would be off 1 second.

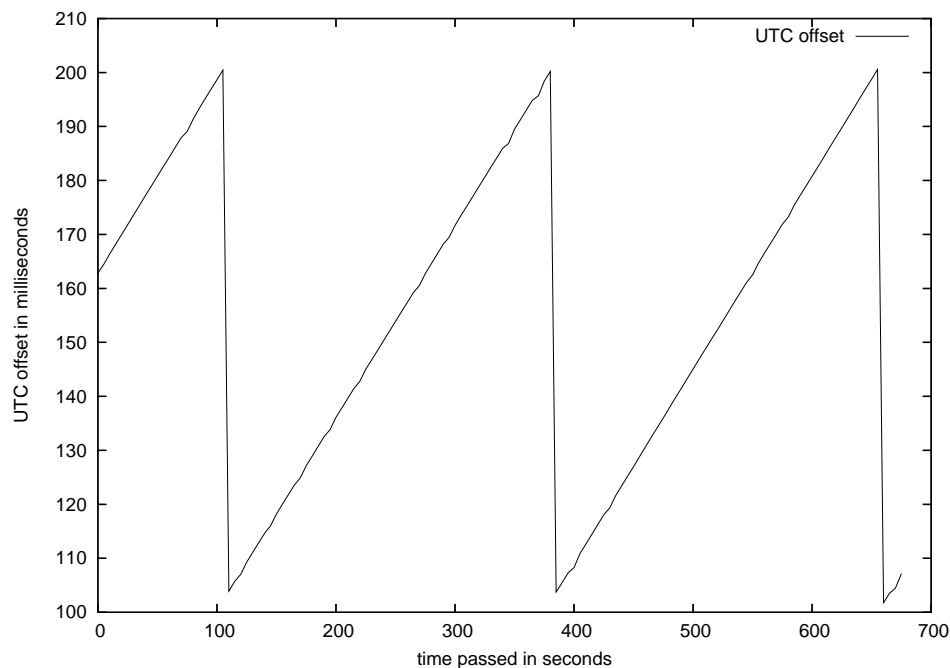


Figure 5.4: Typical clock skew with time adjustment

You can see the clock adjustment taking place when the threshold of 200ms is passed. And you can also see that it does not change its time to zero UTC difference, but to the average of the previous own time and the received time. This is due to lack of confidence or exactness tracking in this simplified time protocol. Still it is very efficient.

In this test the reference clock was a normal PC running ntpd to synchronize his own clock to UTC. The PC used a clock version compiled for the i386 platform

- with just the time sending module (see `sdp/conf/time_server.click` in appendix A).

5.3 Scheduling Updates

Now that all nodes had a synchronized time the scheduling for coordinated switching was added to the activation shell script. It calculates the difference between current time and activation time (from meta-info file). Doing one long sleep then is the most simple, but with clocks drifting so badly this might cause some variance in the switch-over timing. Thus a loop was used that does shorter sleeps and re-checks thereafter, taking advantage of time adjustments from the simple but effective BRN-NTP time synchronization. Because click runs single-threaded and thus blocks until the executed activation script returns, the activation script goes into background when called for scheduling. This worked in simple laboratory test setups, but can be improved in future work.

5.3.1 Test Versions

BRN strives to become a testbed for new software (e.g. routing protocols). For this the notion of testing versions, that would only run for a short time, was introduced.

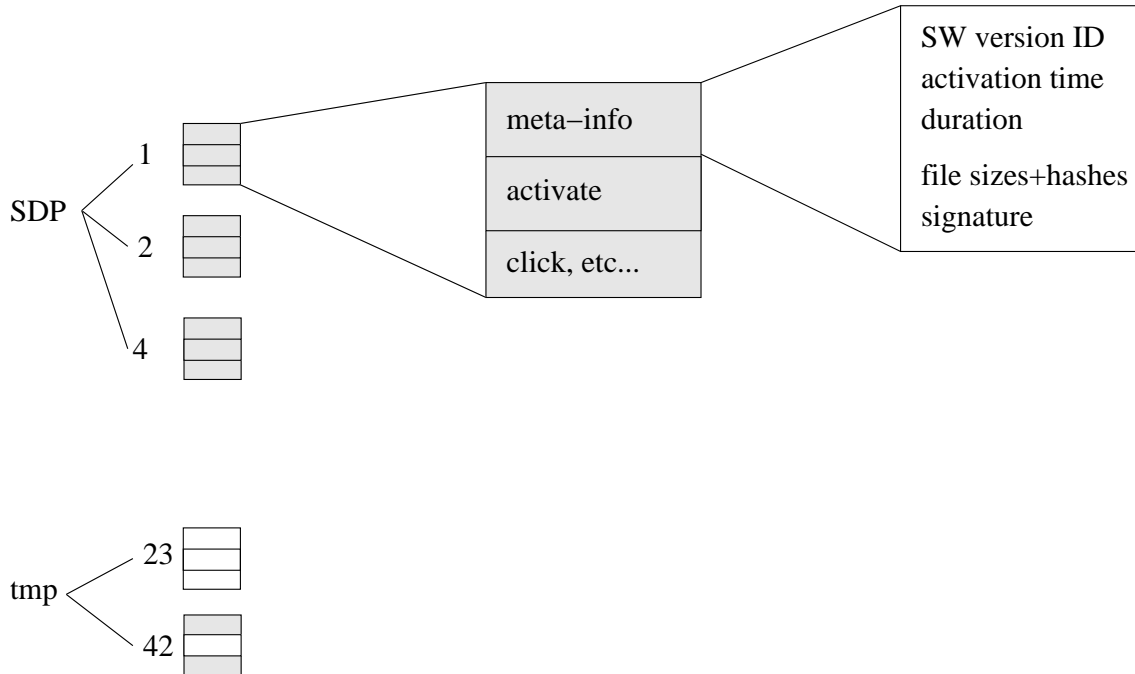
The test software deployment works mostly like regular SDP software updates, including the coordinated switch-over. The only difference being that after testing time expired there will be a coordinated switch-back.

Additionally one might run into space problems for the particular hardware when storing one stable, one experimental and one updated version, but this does not need to be covered here.

Theoretically testing versions might even re-use version identifiers, if after running one version the node switches back to the previous stable version, deletes and forgets the testing version. Of course this would require those versions to be distributed and run in non-overlapping time slots. Also version IDs are not scarce, so the easiest and safest is to always increment the IDs.

5.4 Software Package

This section describes, what constitutes a software package in SDP and explains, why this was chosen.



A software package needs to consists of several files for many reasons. To allow for saving some bandwidth, it is possible to transfer only those files, that are not already on the target node (This is not implemented, yet). Thus the files of a package are not kept as a .tar.gz, but as a collection of files in a directory. If a file is already there, it is hard-linked from its previous location to the new one (not yet implemented - currently all files are always transferred).

Only two of those files are used in a special way by the SDP. One such special file, called “meta-info”, contains a list of all other files, together with their sizes and hashes, followed by a signature. It also stores activation time, activation duration and the version number. If a developer key was used to sign the meta-info, the developer cert would be appended. Another special file, called “activate” is executed at activation time and passed the location of the remaining files, thus the SDP does not need to know anything else about those other files. Just that they all need to be transferred.

A working “activate” shell script was developed and integrated it into SDP. It can can do all required setup like unpacking, sym-linking or hard-linking all files to their proper locations and restarting the routing facilities — that is in our case the click modular router software. The advantage of a shell script is that

it can be easily edited - even with vi on a BRN node for testing purposes thus allowing to considerably ease and speed up development as well as adding to overall flexibility of the SDP system.

The `activate` script receives an action parameter. This can be one of start, stop, restart, status, add, switch, schedule. Here is an explanation of the non-obvious actions:

An **add** will intelligently move a package from RAM to it's final destination, ensuring near atomicity in this process.

A **switch** will immediately switch to a new software version. This includes removing old versions in the course of disk space management.

A **schedule** will do a switch after waiting for the activation time to be reached.

A BRN node may store several versions of routing software, but due to its limited hardware this may only be 2 (worst case: one running and one extra version) or 3. In a minimal setup this would be the currently running version and the next version that needs to be switched to. Usually version numbers increase by 1 from one version to the next version with the only exception being experimental versions, that run for a short period - e.g. at 03:00 am to 04:00 am. See chapter 5.3.1 on page 44.

5.4.1 TFTP

A TFTP¹ was implemented, using Ethernet frames instead of UDP for transport. RFC 1350 [TFTP] was used as reference. Why was TFTP chosen for file transfer? It is robust and simple which both fits our requirement 2 for SDP. Standard FTP uses TCP/IP but our nodes possibly do not have proper IP addresses. BRN does not provide for port numbers like UDP which limits the number of connections between nodes to one at a time. Measured throughput was about 100 to 200 kB/s with 100% CPU usage being the limiting factor with user-space client software.

5.4.2 Managing versions

The chosen hardware platform is rather restricted in terms of RAM and persistent storage, which means that in the worst case only two different software versions can be stored on it. This makes managing installed software versions alone quite a challenge. It not only needs to keep the currently running version,

¹the Trivial File Transfer Protocol is datagram based (UDP instead of TCP) and implements simple error detection and recovery

but it needs to store a fallback version for the case of updating to a non-working version. There are many reasons why software would not work, including

- wrong compiler options or version,
- changing certain header files without a complete rebuild,
- linking against incompatible C libraries,
- click configuration file changes which require a certain networking interface to be active.
- and finally at least one newly received version needs to be stored (preferably on the flash memory).

So all old versions need to be removed automatically and possibly only one future version can be kept in the library at any time. Which version to keep or remove is also a question of policy and demand within the BRN project. To save space and thus make development somewhat more flexible, it is planned to hard-link identical files in different software packages. Those duplicates are efficiently identified using a hash-map lookup table with the MD5 hash value of file content used as keys and a FileInfo structure as data. If and only if the hash is the same, files can be hard-linked. The standard filesystem link counter will ensure consistency and free space when the last entry is unlinked (deleted).

The click software allows for modularization. This means, it is possible to keep the same click main binary all the time and only exchange the SDP and BRN modules at need. In combination with the abovementioned hard-linking this can save both disk space and transfer bandwidth.

5.4.3 Storage Considerations

The Journalling Flash Filesystem (jffs2) employed in the OpenWRT system allows for transparent compression and decompression of stored files. For typical binaries it achieves a ratio of 1:2.7 which is about 70% of what is possible with gzip maximum compression (1:3.8). On the downside with gzip, it is necessary to uncompress the .gz file before you can run it. This will permanently occupy additional memory (about 3MB) but is worth the (about 350k) saved space.

5.5 Analyzing the Code

There is a working basic implementation of SDP consisted of 1500 lines of C++ code and 150 lines of shell script.

This simplicity is deliberate. It is only possible because click provides a well tested and fairly complete infrastructure for handling packets in modules.

Other studies like [Basili1984, Khosh1990] have shown that the number of software errors (bugs) contained in a certain software corresponds to its complexity. Both modularization and small code make the SDP software less complex and thus less error-prone.

5.6 Testing and Debugging

Manual testing can be quite time consuming and unreliable, if you do not exactly reproduce the input behavior.

Click is a mature platform for development of routers. Therefore it provides a useful mechanism called "handlers" to allow for easy debugging and live manipulation of a running router.

There are two types of handlers: read handlers and write handlers. While the latter receive a string from the outside and do something based on it, the read handlers do the opposite. When triggered, they output a string about the element's current state.

For debugging purposes the read handlers can be used to monitor the current state of the router, while the write handlers are useful to change variables at a certain time - e.g. to simulate error conditions.

The click software comes with an element named PokeHandlers. This element can be used to trigger handlers of other elements at certain times.

With some properly integrated handlers in my own modules it is possible to reliably simulate fault conditions like a broken link during different phases of the TFTP transfer.

This also allows for automatic testing of all functionality after every change or before a checkin into the repository. Such automated testing has well known benefits on software quality. However this is outside the scope of this work to discuss.

Additionally it is useful to create a click configuration that includes more than one instance of the components one wants to test, but without **FromDevice**

and **ToDevice** elements. Click can then run operation without access to a real network interface. An example of such a file is given in appendix B on page 57 and graphed in figure 5.5.

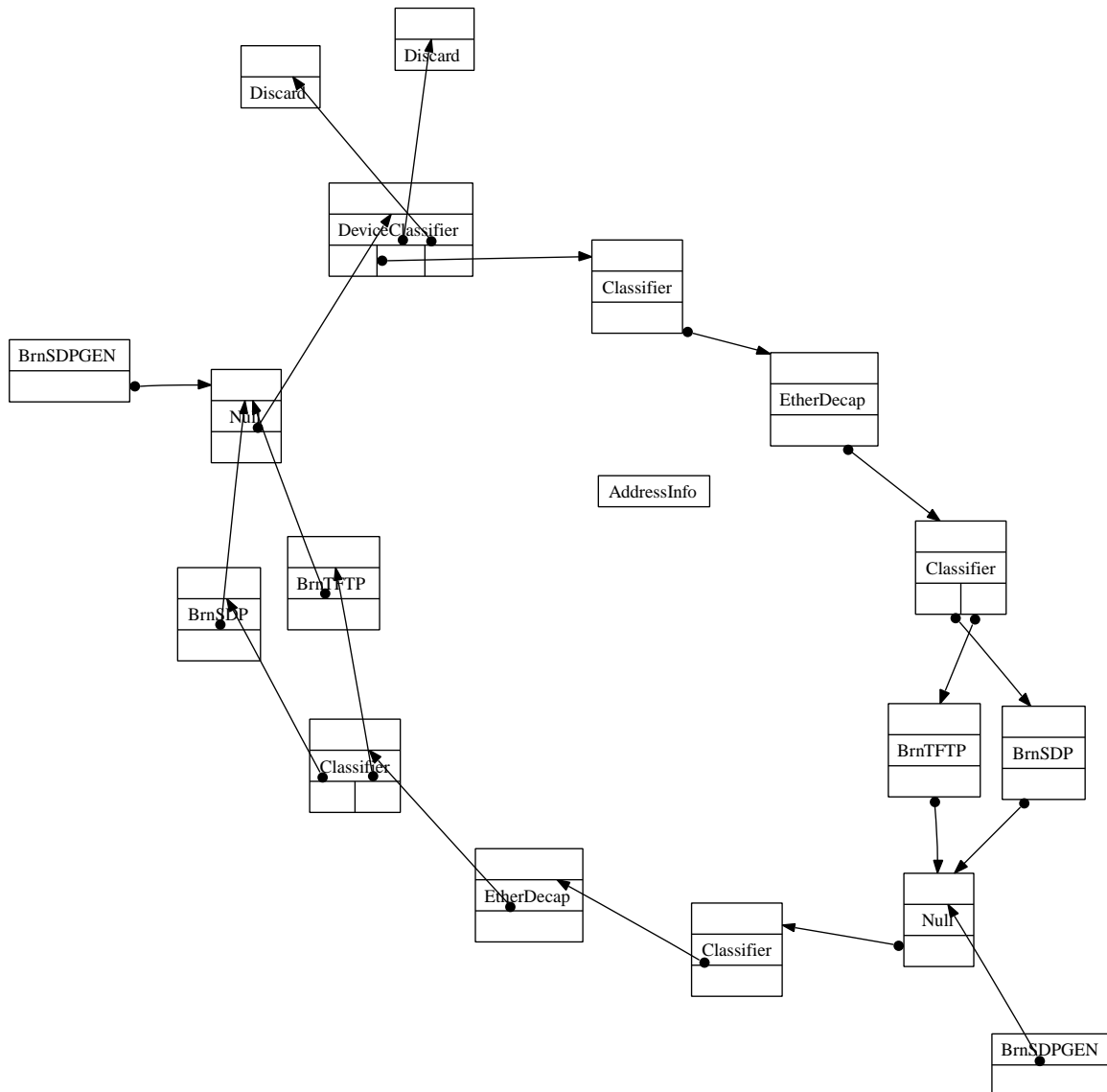


Figure 5.5: Click graph for testing without network

5.7 Scenarios

To better illustrate the working of SDP this section will discuss a few typical scenarios

- Normal operation
- Normal update process
- Update with a disconnected node
- Update with a disconnected subnet

5.7.1 Normal Operation

During normal operation each node sends SDP beacons containing its current software \rightarrow *version ID* and current time in milliseconds.

All nodes run the same software version, hence they do not need to react on the version part of the SDP beacon. They only adjust their clocks from the NTP portion to synchronize with each other. Since some nodes are synchronized from external \rightarrow *UTC* time sources, the whole BRN has their clocks running on (or at least close to) UTC.

5.7.2 Normal Update

For a normal update a new version is injected at any node (or even several nodes, if they are accessible) - let us call this one node 1. This node 1 then broadcasts beacons containing the incremented version ID to its neighbors.

Thus neighboring nodes notice that there is a new version and start by requesting the meta-info file from the node 1 with TFTP. After the signature on the meta-info file is verified, nodes continue to download all other files named in the meta-info file - one by one. After each transferred file the hash value is checked. If the received file was corrupt, it would be discarded and re-transferred to ensure software integrity.

When all files of the new software package are completely transferred, it schedules the software switch to happen at the activation time (known from meta-info file) and starts broadcasting the new version in its own SDP beacons - acting identical to node 1.

A normal update results in all nodes returning to normal operation with the new version.

5.7.3 Update with a Disconnected Node

This discusses what happens when one node of the BRN is disconnected or turned off during a software update.

The update of all other nodes works the same as with a normal update, but the one disconnected node remains at its old version. When it is reconnected, it will send the old version ID in its beacons. This causes all neighboring nodes to reply with their higher version ID.

In response to the higher ID, the previously disconnected node will fetch all files as with a normal update. The scheduling code will notice if activation time is in the past and will immediately switch over to the newly received version. Afterwards normal operation can continue.

5.7.4 Update with a Disconnected Subnet

This case is the most problematic one. It discusses what happens when the BRN is split into two or more parts when some of the central nodes are disconnected.

Like with a normal update, new software would be injected into one of the sub networks and spread within it. Because there is no way to reach the other parts of the BRN, spreading would be limited to the subnet of injection and other subnets would remain at the old versions. Also — independently of SDP — each BRN subnet would only be able to route normal traffic within the bounds of their subnet.

When the subnets get reconnected the new version would spread throughout the newly extended BRN and be scheduled for activation. In the worst case the activation time is in the past and nodes would switch over one after another without the usual synchronization. However, once the new version has propagated (about 1 minute for each hop), the whole BRN will be back in normal operation state.

5.8 Scalability

With a growing network and more users the performance of centralized distribution methods usually decreases. It is interesting to think how a bigger BRN mesh would affect SDP itself.

It should not make a big difference as propagation time is proportional to maximum path length. The maximum path length is approximately proportional to

the diameter of the network. This should make propagation time proportional to square root of number of nodes (not even taking into account that the mesh will get better connected with more nodes). Concluding SDP can be expected to scale well.

5.9 Effort Estimation

Often with software engineering it is of special interest what part is most time consuming. Thus the interested reader is given a rough estimation of how much time went into which tasks.

- 3 weeks understanding BRN general concepts and doing raw/preliminary SDP design
- 3 weeks helping with OpenWrt/OpenWGT setup, development, and integration
- 5 weeks understanding the click modular router framework enough to send and receive a single packet
- 3 weeks designing SDP structures, file formats, etc
- 2 weeks implementing SDP beacon generator and parser
- 2 weeks implementing TFTP on BRN with robust error handling
- 1 week designing, implementing and testing the simplified BRN-NTP
- 2 weeks integrating SDP build and deployment into BRN and OpenWGT (with tests)
- 2 weeks adjusting to the latest BRN platform or subversion structure changes (about 6 times total)
- 8 weeks writing down this thesis in LaTeX
- 1 week printing and binding the physical paper

Chapter 6

Conclusions

6.1 Summary

All basic SDP functionality is designed, implemented and working reliably. The resulting SDP protocol is able to remain cross compatible¹, it is robust — e.g. it recovers from lost packets — it does maintain consistent versions within BRN, making use of the builtin time synchronization protocol and finally it is designed to be secure by employing cryptographic hashes and signatures for software updates.

Many existing methods were be applied to solve this problem and some new ideas have been developed for the special requirements of BRN.

The more advanced — but not mission critical — features are not yet implemented, but easily feasible with some additional weeks of effort.

With this SDP it finally does become possible to deploy BRN nodes on Berlin roofs.

6.2 Future Work

What is improvable?

- Integration of SDP into kernel click to improve performance. This requires wrapping a few functions like `gettimeofday` and `system` so that they can be used in kernel-space.
- design SDP with multicast file transfers to further minimize network usage

¹as long as BRN packet format is not changed again

- better fallback to telnet mode with bug-fixed atheros → *WLAN* driver
- MD5 checks on startup to further improve reliability

In addition to chapter 5.3 on page 44:

The click framework provides for asynchronously running tasks and timers. This can be utilized to schedule future software updates. It is important to recheck once in a while because the local clock (which drifts heavily) might have been adjusted to the network timing.

Using timers it is then also possible to cancel updates that are already scheduled. This would be useful when another software package arrives and is to be activated at the same time (instead of the other one).

Appendix A

BRN NTP

A.1 Example UTC Time Sources

Broadcasting time from a PC

```
sdp/conf/time_server.click:

# load the required module
require(sdp);

# send 4 time beacon per minute:
gen :: BrnSDPGEN(15);
AddressInfo(my_vlan eth0:eth);
gen -> to_dev_clf :: DeviceClassifier(my_vlan, my_vlan, my_vlan);

    to_dev_clf[0] -> out_q_0 :: Queue();
    to_dev_clf[1] -> Discard;
    to_dev_clf[2] -> Discard;

out_q_0
-> ToDevice(eth0);
```

UTC time synchronization on BRN nodes

```
/usr/sbin/ntp-client:

#!/bin/sh
while sleep 100 ; do
```

```

rdate 217.172.177.32 && \
echo 1 > /var/updatelink/disablebrnntp || \
echo 0 > /var/updatelink/disablebrnntp
done

```

A.2 NTP Implementation

excerpt from `sdp/src/brnsdp.c`:

```

/// this time is long past - if we still are before it,
/// our own time is bad - in theory we could also use __TIME__
#define mingoodtime 1083191508000000LL
/// time values before this are invalid
#define minvalidtime 946900000000000LL
/// microseconds network transfer time - 500us=0.5ms for eth
#define networklatency 500
/// minimum offset to step clock in microseconds
#define ntpthreshold 200000

...

if(!_disable_ntp) {
    int64_t peertime=(int64_t)tv.tv_sec*1000000+tv.tv_usec+
        networklatency;
    gettimeofday(&tv, NULL);
    int64_t owntime=(int64_t)tv.tv_sec*1000000+tv.tv_usec;
    int64_t midtime=(peertime+owntime)>>1;
    int64_t time_offset=peertime-owntime;
    if(owntime<mingoodtime && peertime>mingoodtime)
        midtime=peertime;
    tv.tv_sec =midtime/1000000;
    tv.tv_usec=midtime%1000000;
    if(peertime>minvalidtime && abs(time_offset)>ntpthreshold) {
        settimeofday(&tv, NULL);
    }
}

```


Appendix B

Click Configuration Files

A click configuration file with only SDP

```
require(sdp);
require(brn); # for EtherDecap

AddressInfo(my_vlan eth0:eth);

gen :: BrnSDPGEN(5);
sdp :: BrnSDP(gen, tftp, /tmp/sdp);
tftp :: BrnTFTP(sdp, /tmp/storage);

to_dev_clf::DeviceClassifier(11:11:11:11:11:11,my_vlan,my_vlan);
proto_clf :: Classifier(0/0002, 0/0001) // TFTP, SDP

FromDevice(wlan0)
-> Classifier(12/8086)
-> EtherDecap()
-> proto_clf;

proto_clf[0] -> tftp;
proto_clf[1] -> sdp;
tftp -> to_dev_clf;
gen -> to_dev_clf;
sdp -> to_dev_clf;

to_dev_clf[0] -> Queue() -> ToDevice(wlan0);
```

```
to_dev_clf[1] -> Queue() -> ToDevice(vlan1);
to_dev_clf[2] -> Queue() -> ToDevice(vlan2);
```

A config for testing SDP without network

```
require(sdp);
require(brn);
```

```
AddressInfo(my_vlan eth0:eth);
```

```
gen :: BrnSDPGEN(5);
sdp :: BrnSDP(gen, tftp, /tmp/sdp);
tftp :: BrnTFTP(sdp, /tmp/storage);
```

```
gen2 :: BrnSDPGEN(6);
sdp2 :: BrnSDP(gen2, tftp2, /tmp/sdp2);
tftp2 :: BrnTFTP(sdp2, /tmp/storage);
```

```
out_queue :: Null();
in_queue :: Null();
out_queue2 :: Null();
in_queue2 :: Null();
```

```
proto_clf :: Classifier(0/0001, 0/0002) // SDP, TFTP
proto_clf2 :: Classifier(0/0001, 0/0002) // SDP, TFTP
```

```
in_queue
-> Classifier(12/8086)
-> EtherDecap()
-> proto_clf;
```

```
in_queue2
-> Classifier(12/8086)
-> EtherDecap()
```

```
-> proto_clf2;
```

```
proto_clf[0] -> sdp;
```

```
proto_clf[1] -> tftp;
```

```
tftp -> out_queue;
```

```
gen -> out_queue;
```

```
sdp -> out_queue;
```

```
out_queue -> to_dev_clf::DeviceClassifier(11:11:11:11:11:11,my_vlan,my_vl
```

```
to_dev_clf[0] -> out_q1_0::Null();
```

```
to_dev_clf[1] -> Discard();
```

```
to_dev_clf[2] -> Discard();
```

```
proto_clf2[0] -> sdp2;
```

```
proto_clf2[1] -> tftp2;
```

```
sdp2 -> out_queue2;
```

```
gen2 -> out_queue2;
```

```
tftp2 -> out_queue2;
```

```
# connect both virtual nodes:
```

```
out_q1_0 -> in_queue2;
```

```
out_queue2 -> in_queue;
```


List of Figures

| | | |
|-----|--|----|
| 1.1 | BRN from a user's point of view | 8 |
| 1.2 | BRN from the internal network point of view | 9 |
| 3.1 | Overview of properties of software update methods | 13 |
| 3.2 | Netgear Flash and Firmware layout | 23 |
| 5.1 | Overview of the SDP components and their interaction | 37 |
| 5.2 | Click graph of a BRN-NTP time-server | 38 |
| 5.3 | The SDP click graph | 39 |
| 5.4 | Typical clock skew with time adjustment | 43 |
| 5.5 | Click graph for testing without network | 49 |

Bibliography

- [ClickDoc] Click Modular Router documentation -
<http://pdos.csail.mit.edu/click/doc/>
- [Roofnet] J. Bicket, D. Aguayo, S. Biswas, R. Morris "Architecture and Evaluation of an Unplanned 802.11b Mesh Network", 2005, M.I.T. Computer Science and Artificial Intelligence Laboratory
- [Netgear] Netgear, <http://www.netgear.com/>
- [Trid1999] A. Tridgell "Efficient Algorithms for Sorting and Synchronization", Australian National University, 1999
- [HTTP] <http://www.ietf.org/rfc/rfc2616.txt>
- [FTP] <http://www.ietf.org/rfc/rfc0959.txt>
- [TFTP] <http://www.ietf.org/rfc/rfc1350.txt>
- [CVS] <http://www.nongnu.org/cvs/>
- [NTP] <http://www.ietf.org/rfc/rfc1119.txt>
- [TimeProt] Time Protocol <http://www.ietf.org/rfc/rfc0868.txt>
- [SHA1] NIST standard <http://ietf.org/rfc/rfc3174.txt>
- [W3C1997] W3C, "The Open Software Description Format"
<http://www.w3.org/TR/NOTE-OSD>, 1997
- [Perc2003] C. Percival "An Automated Binary Security Update System for FreeBSD", 2003
<http://www.daemonology.net/freebsd-update/binup.html>
- [WhIr2004] D. White, B. Irwin "Microsoft Windows server update services (WSUS) review", 2004

[Tall1995] Owen H. Tallman "Automated Software Deployment in a Large Commercial Network", Digital Technical Journal, 1995

<http://www.hpl.hp.com/hpjournal/dtj/vol7num2/vol7num2art5.pdf>

[GhRo2005] C. Ghantsidis, P. Rodriguez "Network Coding for Large Scale Content Distribution", IEEE Infocom 2005

http://www.research.microsoft.com/~pablo/papers/nc_contentdist.pdf

[Cohen2003] B. Cohen "Incentives build robustness in bittorrent", Workshop on Economics of Peer-to-Peer Systems, 2003

[RSA1978] R. Rivest, A. Shamir, L. Adleman "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, 1978

[Basili1984] V. Basili, B. Perricone "Software errors and complexity: an empirical investigation". 1984

[Khosh1990] T. Khoshgoftaar, J. Munson "Predicting software development errors using software complexity metrics", 1990

Erklärung

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den November 14, 2005, Unterschrift

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den November 14, 2005, Unterschrift

Fair use allowed

General permission to make fair use in teaching or research of all or part of this material is granted. ©2005 Bernhard M. Wiedemann