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

# Diplomarbeit

Thema: Entwicklung einer Softwareverteilungsplattform für das Berlin Roof Net

Topic: Development of a Software Distribution Platform for the Berlin Roof Net

eingereicht von: Bernhard M. Wiedemann
Betreuer: Prof. Dr. Jens-Peter Redlich

Berlin, den 29.11.2005

# Acknowledgements

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

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

Last but not least, I also need to thank my family for their patience and understanding in letting me research and write for so many hours without having time to play with my son or to discuss philosophy with my beloved wife.

# Contents

# Abstract

The 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.

In the course of this work 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.

## Zusammenfassung

Das Berlin Roof Net (BRN) Projekt hat zum Ziel, ein dezentrales, drahtloses, multi-hop Maschennetz aufzubauen, welches Gegenstand aktueller Forschung ist. Es gibt zahlreiche Maschennetz-Routingprotokolle mit verschiedenen Stärken und Schwächen sowie Parametern, um Datendurchsatz, Latenz, Zuverlässigkeit, Sicherheit usw. zu verbessern. Die dazu nötige neue Software in diesem dezentralen Netz ohne IP-Adresszuweisung und ohne Routing zu verteilen, stellt allerdings eine beträchtliche Herausforderung dar. Dazu bedarf es einer wohl durchdachten Softwareverteilungsplattform (SDP). Diese Diplomarbeit umfasst den Entwurf und die Umsetzung einer solchen SDP.

Es wird dabei gezeigt, dass eine zuverlässige Softwareverteilung mittels viraler Verteilung über Direktverbindungen zwischen den BRN-Knoten möglich ist.

# 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 whether 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 them 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.

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.
A →*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[1] 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[2] and used in the Roofnet project[3].

[1] `http://sar.informatik.hu-berlin.de`
[2] `http://pdos.csail.mit.edu/click/`
[3] `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. The CA is signing developer or software package 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 (formerly known as GMT): the time all atomic clocks run on — with some added leap-seconds. 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. Versioning will be explained in chapter 4.2.4 on page 31.
- *Meta-info*: This denotes all the information that is stored about a →*package* and transmitted in the `meta-info` file, including →*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 5.3 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, an overview of the current state of the art will be given 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, insight into the actual code, development process, measurements, and other details about the SDP implementation is given.

## 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 details some aspects 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 set up much easier than normal networks.

Simple ad-hoc wireless networks have existed 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 a Software Distribution Platform essential.

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

- It has no privileged nodes,
- It has no centralized resources (e.g. data-store),
- It runs on homogenous nodes (same hardware and software) with different administrators,

- Nodes may be turned off or disconnected from the remaining network for any period of time,
- The network may be split into several subnetworks for any period of time,
- Nodes may be added at any time.

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

- Nodes are installed on roofs and other locations that are difficult to reach,
- 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 →*WLAN* access points for IEEE 802.11g standard,
- BRN wants to become a community project/network,
- Nodes are set up from inexpensive and thus limited hardware (e.g. the Netgear WRT54GS router 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 one can not easily limit what applications can do. It is even harder to do so on limited hardware like the one of BRN nodes.

**Why was the BRN started?**

People nowadays like having Internet everywhere. Of the methods available to get access to the Internet from laptops or handheld devices, it is wireless technology that is the most prevalent and convenient to use. So what people want, is to 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 them connect, they need a base station with Internet access. 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 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 commercial 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.

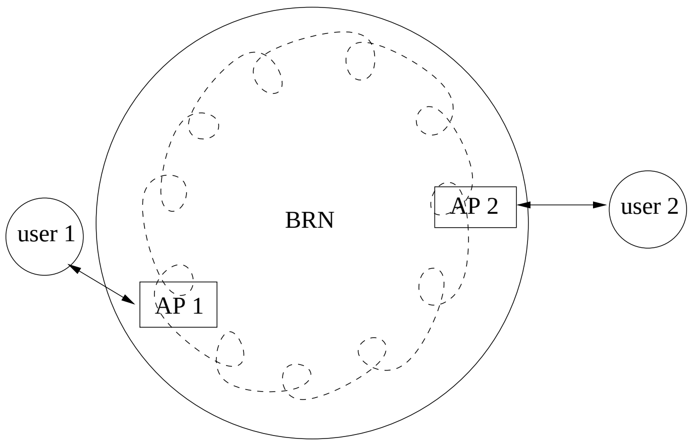



Figure 1.1: BRN from a user's point of view

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

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 →*DHCP* server would be such a resource.

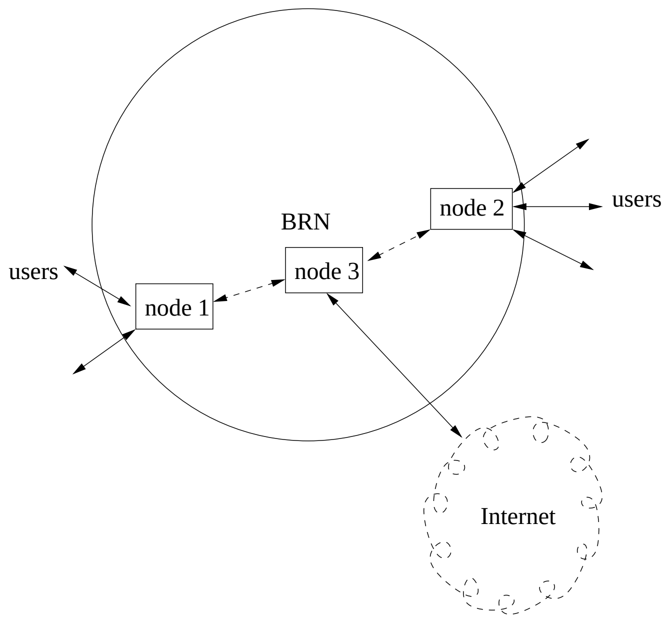



Figure 1.2: BRN from the internal network point of view

### Hardware

Because the BRN project aims to cover a vast area with its wireless mesh network, it needs many inexpensive nodes to establish it. Thus commercial off the shelf routers with integrated access-points are used. At an earlier stage Linksys WRT54GS systems were used and nowadays BRN uses Netgear WGT634U systems.

Not surprisingly they contain hardly more hardware than is needed for the job they normally do: 8MB flash, 32MB RAM, a 200 MHz MIPS CPU, a 5-port Ethernet interface and a 802.11g compatible WLAN chip which is working with the MADWIFI Atheros open source driver.

### Software

A modified OpenWrt distribution is used for the base system. It is an open source distribution for the Linksys WRT54G platform. OpenWrt employs the Linux kernel, the `uClibc` C library, and many space-saving `busybox` utilities together with more specialized tools like `iptables` and `iwconfig`. OpenWrt has been ported to the Netgear WGT634U platform by other BRN project participants. The resulting distribution is sometimes called "OpenWGT" but is largely compatible to OpenWrt thus the terms are interchangeable for the purpose of this work.

On top of OpenWrt runs the click modular router software that allows to capture, process and produce packets using reusable filter modules. The click software is explained in chapter 5.1 on page 38.

# Chapter 2

# Problem Statement

This thesis is about the design of a Software Distribution Platform (SDP) for mesh networks in general and it includes a basic implementation for →*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:

- Building update packages
- Notifying nodes about software updates
- Transferring files
- Verifying package integrity and authenticity
- Synchronizing system clocks
- Switching software versions

This work will focus on providing the basic functionality so it can be built upon. Any employed method can be optimized in future work.

While security is generally included in the discussion, it is not the primary concern. Furthermore mesh-networks, routing, and the OpenWrt/OpenWGT system hardware and software are only slightly covered to better show, how SDP integrates into it.

#### An Example

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

In an initial stage, 20 nodes might be set up 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. Operating with mixed versions could cause network partitioning which means that it is impossible to reach a part of the network. With non-functional software it is even likely that nodes are left without routing which complicates access to correct the software.

# 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 selected existing methods to improve the reader's understanding about differences and similarities. The next chapter will then discuss to what extent current methods can be employed to meet specific requirements of SDP.

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 original 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 the following formats:

- As complete source code package
- As source code differences or patches
- As complete 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. updating original Netgear access-points.

Those methods follow common principles and steps:

1. Software is packaged into a common well defined →*package* format and tested prior to delivery.
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 may include executing the pre-install and post-install scripts from 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 may include executing the pre-uninstall and post-uninstall scripts from the old package.

The order of steps 4, 5, and 6 may vary.

**Software Distribution with P2P**

Apart from the aforementioned there are also peer-to-peer (P2P) techniques like Avalanche and BitTorrent [GhRo2005, Cohen2003] that can be used to distribute software. 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. Most real world P2P applications rely on IP and routing.

#### Software Distribution with Worms

Worms (sometimes mistakenly called "viruses") distribute themselves. They often distribute without user interaction and often spread very rapidly throughout Internet hosts. The general common principle for the distribution of worms is that once a host becomes infected by a worm, it starts sending out worms itself to other reachable hosts without requiring any centralized resources. An example is the Sapphire/Slammer worm that was analyzed in [Sapphire]. It demonstrated that it was possible to infect 90% of vulnerable hosts within 10 minutes. This allows to conclude that infection based distribution methods work well.

The difference to regular software updates is of course that computer worms are malicious programs that enter the system through unintended channels and often cause 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.

Notification of updates can be done in various ways, including sending a broadcast packet, or by pushing data over a TCP/IP socket. E-mail notification is common but apart from this there seem to be no established standards.

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 are 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 is possible 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 safe storage space and transfer bandwidth. Examples are: deb (Debian GNU/Linux), rpm (Red Hat/SUSE/Mandrake Linux), msi (MS Windows), zip, tar.gz (generic file collection)

To finish this section, a rough summary is given below.

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 |

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.

[1] push 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.

[2] not by itself but with binup [Perc2003]

[3] Updates can only be decentralized insofar as a local mirror can be set up anywhere — usually in large administrative entities — and afterwards be run independently from the vendor.

## 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 Red Hat 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 rpm format. It queries the rpm database about the `rpm` package's meta-information: **`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
Install date: Di 18 Okt 2005 13:29:37  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, Key ID a84edae89c800aca
Packager    : http://www.suse.de/feedback
Summary     : The RPM Package Manager
Description : RPM Package Manager is [...]
```

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 provided to solve security problems, but sometimes updates address other flaws that could render the software or the whole 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 that watches various sources about newly known vulnerabilities. They classify the threat potential based upon 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 update packages with pre-installation information or just use the complete list instead.

In the next step YOU downloads update packages. 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 package 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 gives 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. Gentoo Linux uses the `emerge` utility to easily manage all package-related operations.

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.

Note that this is very different from most other commonly used software distribution methods in that only platform independent source code is provided and compiled for the target platform with user-defined options on demand. It also allows users to customize packages with various compile time options.

This is covered in the comprehensive Gentoo documentation that can be found online at [Gentoo].

### 3.3.2 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 and patches.

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 packages[4].

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

### 3.3.3 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. Alternatively firmware images could be shipped on CDs. This firmware image is then manually uploaded to the embedded devices. All three devices mentioned later in this section run different versions of embedded Linux.

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

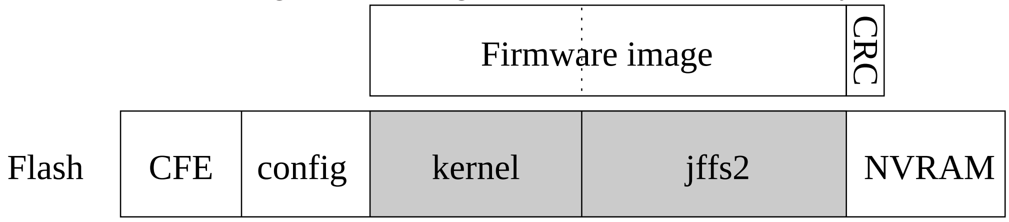
The AVM Fritz Box update packages are `tar` archive files that are uploaded through HTTP to the device, after password authentication. Once the file is completely transferred, the archive is unpacked into RAM and the `install` script is executed. The usage of the remaining content of the package is solely determined by the installation script. AVM commonly releases updates with a `filesystem.image` using squashfs and a `kernel.image` binary that are then written to flash memory.

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

#### 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 (Journalling Flash File System, version 2) 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.

Figure 3.2: Netgear Flash and Firmware layout




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 (Non-Volatile Random Access Memory) that stores all settings for the CFE like MAC address, kernel location or bootloader options.

#### 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 ROM filesystem (cramfs) is used that can be efficiently decompressed on demand by the Linux kernel.

### 3.3.4 Current Mesh Network Updates

There are a few working mesh networks operating in the world and even in Berlin, so it is interesting to know how they distribute and install new software.

#### freifunk.net

Like some others `freifunk.net` is employing the OLSR (Optimized Link State Routing) protocol to build a wireless mesh network. Firmware images can be freely downloaded by any user from the web-site and has to be installed manually. Sometimes this process 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 it was stated 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. Most requirements for SDP arise from there.

The SDP must provide reliable and backward-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 temporary 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 29.

Additionally, SDP has to be backwards-compatible — beginning with the first released version (let us call it "version 0" ; see chapter 4.2.4 on page 31 on versioning) to allow upgrading a very old node to the most recent version. This requires the network protocol to remain stable. If this were not the case, updating old/broken nodes would pose an enormous problem[1].

[1]One would first need to locate those nodes which might have been installed on roofs or other locations difficult to reach. Even when a node is in one's physical reach, updating such a broken node, might still pose a problem, because they usually do not have a known IP address assigned

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 4.2.5 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. Maintain a backward-compatible network protocol,

2. Maintain consistent versions,

3. Be robust,

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

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 14, 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, as this would not only require

to any interface, no VGA - only maybe a serial console.

some authority to operate the hardware but would also possibly introduce a single point of failure. 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 those 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 a home network, it is not suitable for BRN because it would require days, or even weeks until the network reaches sufficiently consistent versions. However, in the meantime the network would not be functional.

The comparative table given in chapter 3 on page 13, shows 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 one 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 to avoid running mixed versions within the network. This delayed activation will be detailed in chapter 4.2.5 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 (i.e. file server), which might otherwise become a bottleneck when the mesh grows to hundreds or thousands of nodes, all requesting the new version at the same 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.

Apart from the technical aspects there is also something to be learned from existing policies on updating software. One point is that all non-trivial changes should be tested internally, before distributing them and another point is that interfaces should remain stable, whenever possible.

#### How to distribute

As found earlier, software can be distributed offline, by downloading, and online. Of course the offline method would need days, if not months to spread a new version and would therefore result in differing software versions and thus fail to meet the criterion of version-consistency, which is requirement 2 for SDP. That makes it entirely unsuitable as testbed.

The offline method shares some problems with updates by downloading. Node owners can be away on vacation, be unreachable by e-mail or just be 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 (that is 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. Difference updates can be simplified by employing software that would allow layering directories. Such software is available but not fully functional and stable: the translucency Linux kernel module, written by myself, and the unionfs[2]

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 size 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 building such an update. Additional testing in a small mesh network 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 element is code to transfer and update a complete image for major updates.

### 4.2.2 Robustness

This section deals with requirement 3 from chapter 4.1 on page 25.

Robustness is required because 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 trans-

[2]union filesystem home page: `http://www.fsl.cs.sunysb.edu/project-unionfs.html`

mitting 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, a watchdog script was made 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 defect neighbor node or a broken antenna. 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 which then can try to start alternative software or could enable a telnet daemon, so that remote administration is possible over the wireless interface. This allows for both diagnostics and repair operations 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 receive anything from their neighbors for some time (e.g. because of a 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, thus causing a chain-reaction.

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

SDP's fallback mechanisms do not need to cover failures which are easily avoided by the recommended in-house testing. This includes syntax errors in the activate script and problems with linking and loading click and its modules.

To further improve fallback handling, 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 but this is outside the scope of this work.

### 4.2.3 File Transfer

As mentioned earlier files are distributed like computer worms by an infection method. For file transfers over single hops a variant of TFTP[3] is used, based on the official [TFTP] RFC 1350 but using BRN packets instead of UDP/IP datagrams. See chapter 5.3.1 on page 47 for implementation details.

### 4.2.4 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 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.

As another simplification, downgrading software needs to be done by upgrading to the next version. This way the SDP version management is simplified to fetching versions with a higher ID and switching to them at the right time. Only the meta-info file needs to be transferred for this to work. While this ap-

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

proach might seem strange at first, it makes sense because there is always a reason for changing to an old version, thus it really is an upgrade. The advantage of this method is that it utilizes the normal distribution method to guarantee authenticity, integrity and consistency of versions. The alternative for automatic downgrading would be to implement a secure, network-wide distributed voting algorithm but that is outside the scope of this work.

### 4.2.5 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. As BRN nodes do not have unique IP addresses, this can not be done by standardized protocols like NTP. So a simple custom BRN-NTP protocol was employed for time synchronization using broadcasts. See chapter 5.2 on page 39 for discussion of the BRN-NTP 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 5.2.2 on page 44 for implementation details of experimental updates.

### 4.2.6 Notification

As mentioned earlier, SDP has to work without IP. Thus SDP uses local Ethernet or →*WLAN* →*broadcast* packets (OSI layer 2 as defined in [Zimm1980]) to notify neighbors about the current version identifier. Including other information about updates is not necessary, because it is included in the signed — and thus trusted — `meta-info` file. This file contains all necessary information about a software package for SDP and can be requested from a neighbor using the version ID and the source MAC address. Taking untrusted information into account could pose a problem for security.

SDP utilizes 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 yields a total distribution time of $n \cdot T$, 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 2 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.

To allow several mesh networks to coexist in the same region a unique magic number is included in every notification packet. Every node of the same network should use the same value. Notifications with other values will be disregarded to avoid flooding the other network with invalid meta-info files.

Additionally, when a node delivers incorrect data, new transfers with that node should be prevented for an exponentially increasing time. While this measure largely reduces the impact of accidentally overlapping networks, it does not prevent determined attackers who can use forged MAC addresses to send files.

### 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: nobody shall be able to tamper with the BRN 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. As a result of this shortcoming BRN-NTP can not verify authenticity of time synchronization information.

The OpenSSL package already has all necessary code and is available for the OpenWrt platform. SDP signs the SHA1[4] value of the meta-info file (which contains SHA1 values of all other files). The RSA[5] 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, given the way they are used 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 the SDP code.

Also there is another, much better way to implement indirect signing. This

---

[4] see [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)

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

works by letting 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 CRL updated and consistent on the CA host machine. With this method is even possible to implement more complex policies, e.g. software distribution policy could demand a signed test protocol for a new version to be accepted and the software on the CA machine could 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

DoS stands for Denial of Service. A DoS happens when an attacker prevents regular users from using some resource or service. In the BRN case it is easy to jam the network by repeatedly sending on the same frequency and there is not much that can be done about this kind of 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 approximately 10000-100000 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 temporary 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, one 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 and as second step 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. The dashed ellipse shows components that belong to SDP and were developed in the course of this thesis.

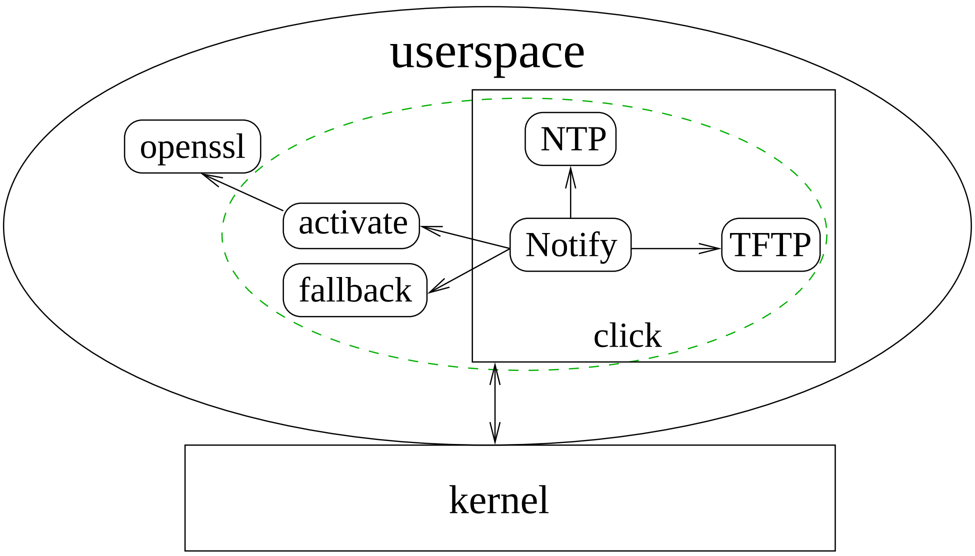



Figure 5.1: Overview of the SDP components and their interaction

The complete SDP source code is provided for reference and for interested readers at `http://bmwthesis.webhop.net`[1]. It is licensed under the GNU General Public License version 2 which explicitly allows use, modification and redistribution of the code as long as the sources are published.

[1]alternatively `http://www.zq1.de/bmwthesis`

## 5.1 Click Modular Router

As 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. Click runs single-threaded with one effect being that only one packet can be processed at a time.

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 on the next page is a visualization of such a graph, created with the `click-viz` and `graphviz` tools from the `time_server.click` file which can be found in appendix A. 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`**. The separate **`AddressInfo`** element does not route packets but is there to store information about a network interface that can be used in other filtering elements.

A more complex click graph with the data flow of all SDP click elements is shown in figure 5.3 on page 40. You find the corresponding configuration file in appendix B.

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

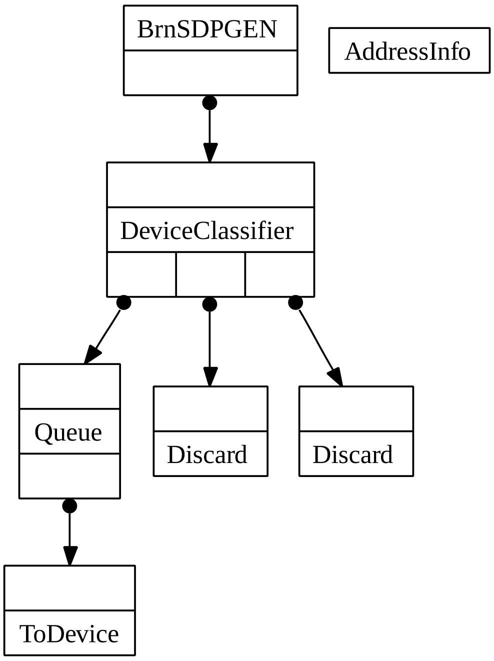



Figure 5.2: Click graph of a BRN-NTP time-server

## 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 time-stamp 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 does not need conversion from and to the gettimeofday and 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.

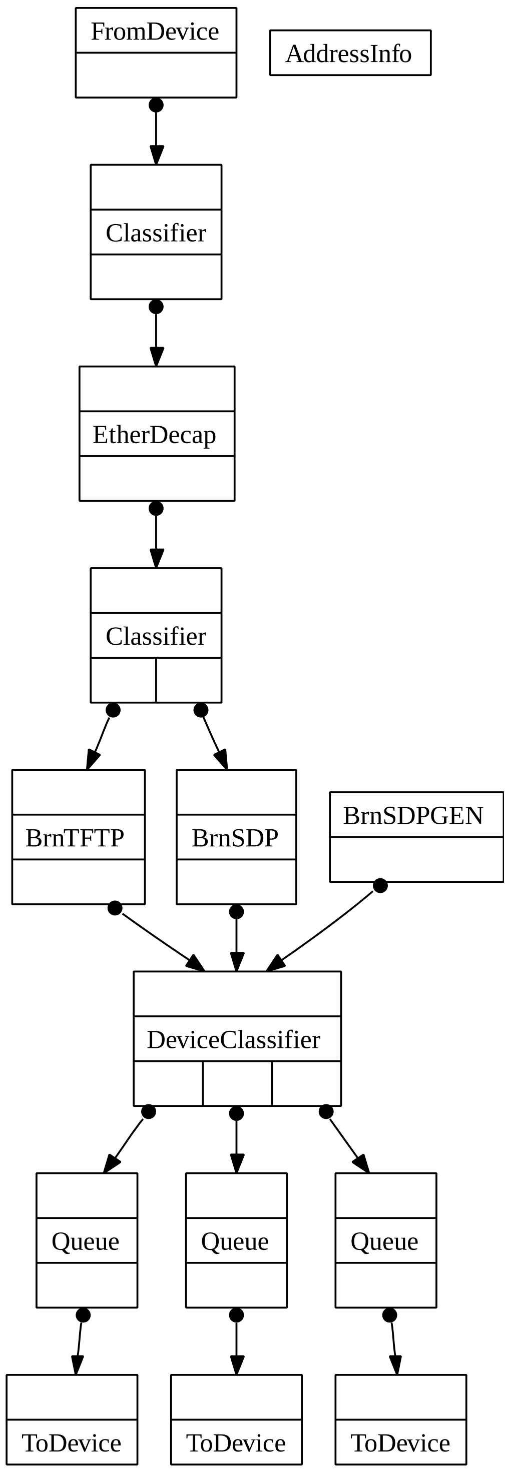



Figure 5.3: The SDP click graph

You can find the actual C code as 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. The effect is even worse when the system time is decreased but this usually does not happen. In this case timers cease to be called until the system clock comes back to its previous value.

This deficiency of the click software adversely affects the fallback timer. It is worked around by changing the fallback script (introduced earlier in chapter 4.2.2 on page 29) 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 examined is slightly different from the actual current implementation but the qualitative result should be the same. The new time of a node is calculated as follows:

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

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, it is assumed that there is 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, nodes 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 $o = \textbf{const}$).

This means that if $o/(n+1) = u \cdot d$ then the two effects will nullify each other. This allows to determine $o = u \cdot d \cdot (n+1)$ The above example values result in $o = 60 \cdot 0.1 \cdot 6 = 36$. 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 neighboring nodes need to be added. Increasing the number of UTC nodes or sending time information more often will also result in better synchronization. In order to improve reliability at least three nodes should have a real UTC time source — preferably in different regions of the mesh network.

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. However, operating entirely without an UTC time-source is not recommended for the BRN, because software updates contain an absolute time for their activation. Only occasionally there might be temporary network splits that could render subnetworks without UTC time. The connected subnetworks will still have synchronized clocks with each other but there is no guarantee as 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.

One can see the clock adjustment taking place when the threshold of 200ms is passed, and one can also see that is 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 its own clock to UTC. The PC used a click version compiled for the i386 platform with just the time sending module. See `sdp/conf/time_server.click` in appendix A.

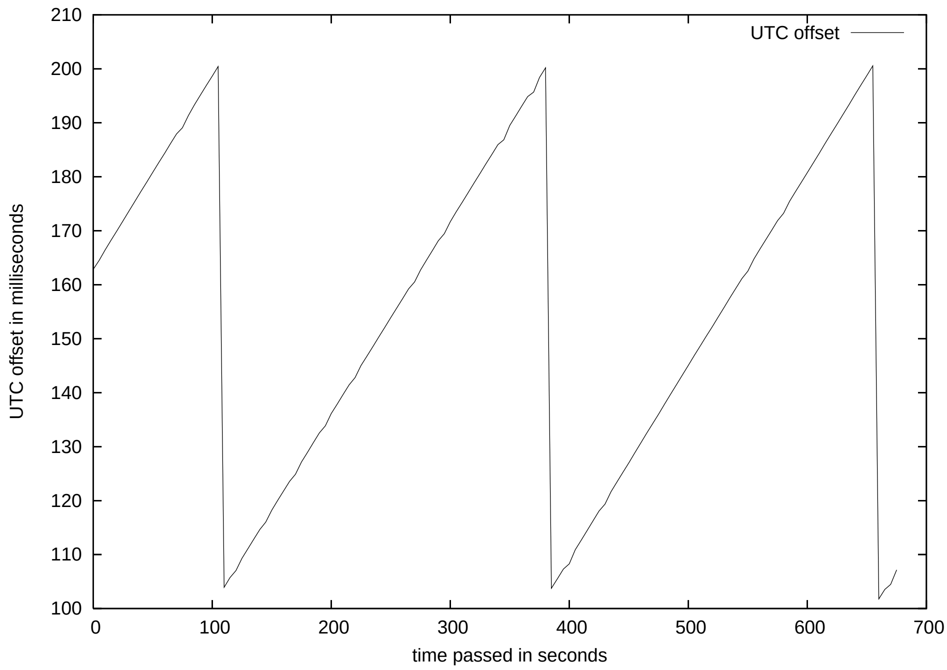



Figure 5.4: Typical clock skew with time adjustment

### 5.2.1 Scheduling Updates

After all nodes had synchronized time the scheduling for coordinated switching was added to the activation shell script which is explained in chapter 5.3 on the facing page. 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.

### 5.2.2 Test Versions

BRN strives to become a testbed for new software (e.g. routing protocols). Therefore the notion of testing versions was introduced.

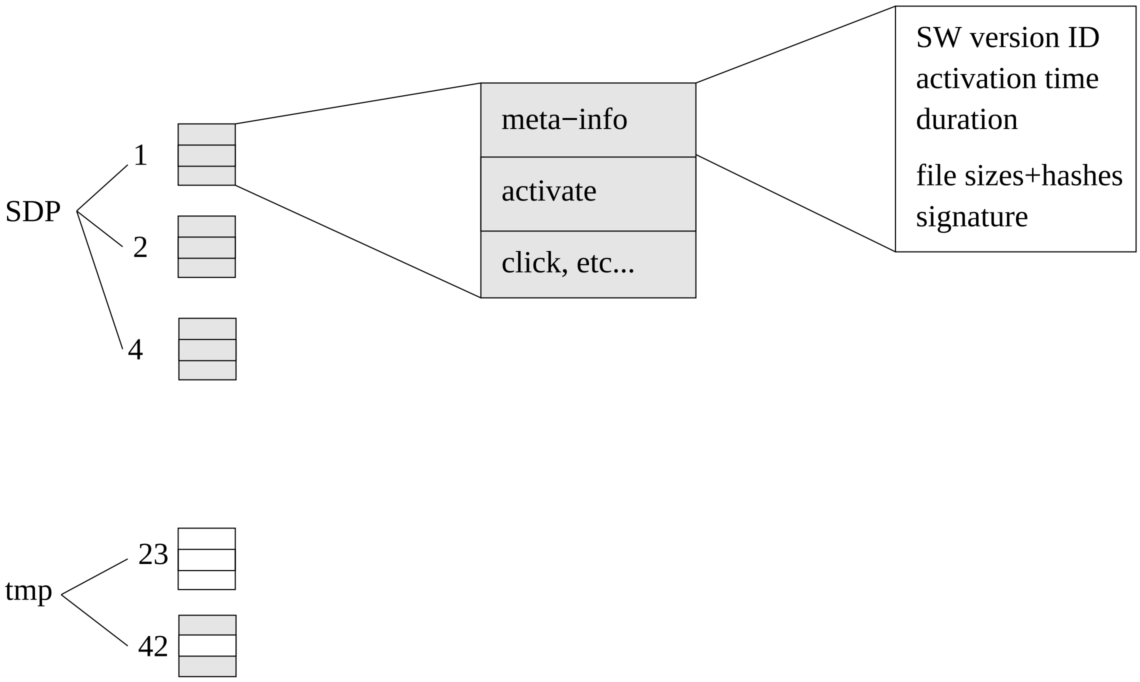
The test software deployment works mostly like regular SDP software updates, including the coordinated switch-over. The only difference being that after testing time expired there will be a coordinated switch-back.

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 because it is implementation specific.

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 way is to always increment the IDs.

## 5.3 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 implemented yet — 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 (to disallow oversized files) and hash values (to verify data integrity), followed by a signature (to verify authenticity). It also stores activation time, activation duration and the version number. If a developer key was used to sign the meta-info, the developer certificate would be appended as well. In the current implementation the meta-info file is limited to a certain size to prevent attackers from filling up a node's RAM. The downside is that this limits the number of files that can be transferred. To allow for an unlimited number of files, the meta-info should be changed to only list one file named "file-list" and that file contains information on all other files (not implemented yet).

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 — see appendix C on page 63. It can do all required setup like unpacking, symlinking 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 its temporary location in the tmpfs to its 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 5.2.2 on page 44.

### 5.3.1 TFTP

A TFTP[2] was implemented using Ethernet frames instead of UDP for transport. RFC 1350 [TFTP] was used as reference. TFTP chosen for file transfer because it is robust and simple which both fits our requirement 3 for SDP. Standard FTP uses TCP/IP but our nodes possibly do not have proper IP addresses. So this TFTP implementation uses BRN packets with an added transport identifier (TID) that is equivalent to the UDP port number in the TFTP on UDP. This TID allows to distinguish between different transfers.

#### Performance

Performance tests have been conducted on a small network of 18 BRN nodes. A package of 1877 Kbyte was transferred approximately 1100 times using different combinations of participating nodes. Tests showed that after injection, all neighbors start and finish downloading at the same time.

Figure 5.5 shows the time needed to transfer the package over a single WLAN hop. The right side shows the corresponding total bandwidth.

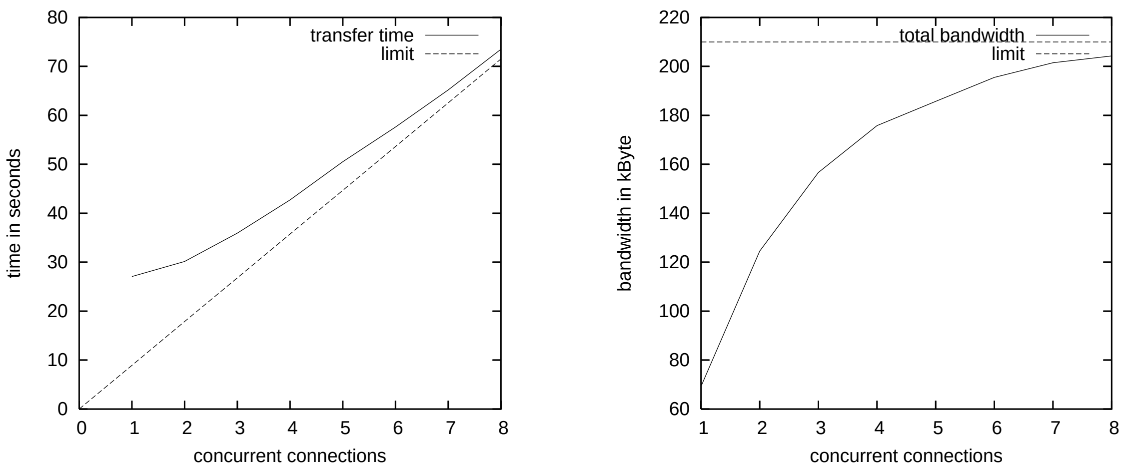



Figure 5.5: Distribution time and bandwidth over one hop

The measured throughput was about 100 to 210 KB/s with 100% CPU usage being the limiting factor with user-space click software. This performance might be improved by utilizing click in kernel mode.

Another limiting factor was the network latency combined with the fact that

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

TFTP waits for an acknowledgment after every single data packet. This could be improved with a more elaborate sliding-window flow control algorithm.

The minimum observed transfer rate was 3Kbyte/s. This can be easily explained by a link with moderate packet loss. The simple BRN-TFTP uses a constant timeout value of currently 500ms and a window size of only 1 packet, as opposed to TCP which sends several packets without waiting for an acknowledgment. So if 25% of packets (2 of 8) are lost, there would be 6 packets per second of 512 bytes each transmitted and 2 timeouts per second resulting in 3Kbyte/s transfer rate.

The total time to distribute a package to all 18 nodes varied between 180 and 260 seconds, depending on which link was chosen for transfer and how many transfers were canceled. Injecting the package took 13 additional seconds.

Concluding, this section has shown that files can be transferred reliably with TFTP but performance can be improved in future work.

### 5.3.2 Managing Versions

The chosen hardware platform is rather restricted in terms of RAM and persistent storage, which means that in the worst case scenario, 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 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.

### 5.3.3 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 it can be run. This will permanently occupy additional memory (about 3MB) but it is worth the saved space (about 350k).

## 5.4 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.

## 5.5 Testing and Debugging

Manual testing can be quite time consuming and unreliable, if one does 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 handers 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 the SDP 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 functions 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 59 and graphed in figure 5.6 on the next page.

## 5.6 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

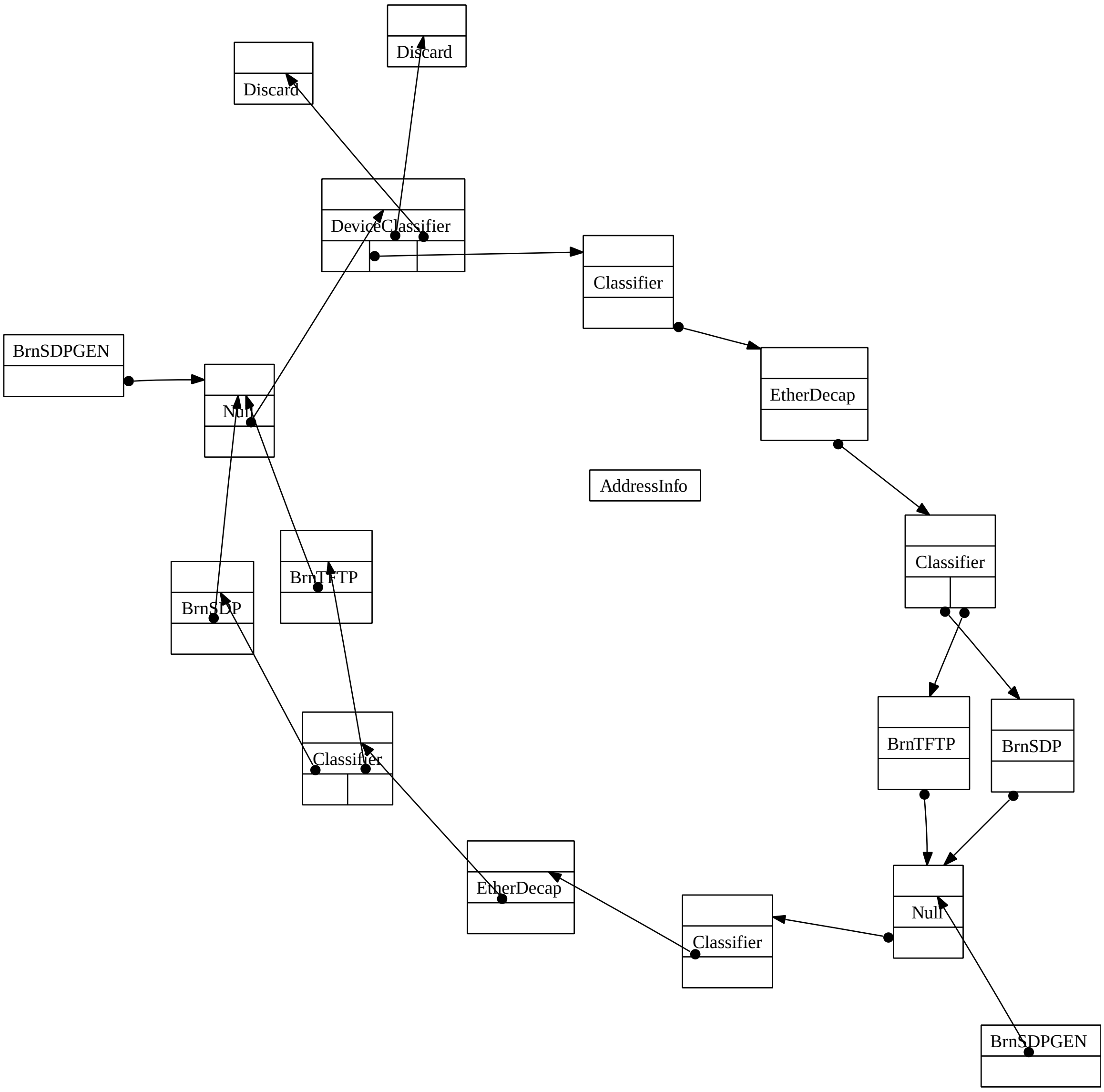



Figure 5.6: Click graph for testing without network

### 5.6.1 Normal Operation

During normal operation each node sends SDP beacons containing its current software →*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 →*UTC* time sources, the whole BRN has their clocks running on (or at least close to) UTC.

### 5.6.2 Normal Update

For a normal update a new version is injected at any node (or even several nodes, if they are accessible) — this one shall be called 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.

### 5.6.3 Update with a Disconnected Node

This section 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.

### 5.6.4 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 subnetworks 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 subnetworks 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.

## 5.7 Scalability

Usually the performance of centralized distribution methods decreases with a growing network and more users. It is interesting to think about 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. The diameter is roughly proportional to the square root of the total number of nodes. 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). Thus, SDP can be expected to scale well.

## 5.8 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 was used for sub-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 including stability and performance 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
- $\frac{1}{2}$ week printing and binding the drafts and final 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 compatible[1], it is robust, 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.

Some existing methods were 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's roofs.

## 6.2 Future Work

There have been a number of ideas for improving SDP that could not be fully researched and implemented, leaving them for future work.

First, SDP should be integrated 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. It has been noted that calling user-space programs from the kernel is unclean design, so a different but compatible asynchronous mechanism should be designed.

[1] as long as the basic BRN packet format is not changed again

Various methods that could further improve reliability should be examined, e.g. running MD5 checks on startup or adding a CRC to BRN-TFTP packets.

Different fallback strategies can be tested, e.g. fallback to telnet mode with a proper Atheros →*WLAN* driver

SDP could transfer its files with local broadcasts to several neighbors at once to minimize network usage. Additionally, transparent compression on file transfer could be added to save bandwidth.

The BRN-NTP accuracy can be improved by using an additional time quality value ("stratum") that is 0 at a UTC node, incremented on every hop, and used to weight time information.

A more general solution to the problem identified in chapter 5.2 on page 41 should be found and implemented. Timers should not be triggered or suspended when changing the local node time but as this is a problem with the underlying click framework it is out of the scope of this work.

Using asynchronously running tasks and timers allows to schedule future software updates while regularly rechecking the local clock. It is then possible to cancel updates that are already scheduled as well.

Updating and scheduling more than one software-update at a time might also be interesting to BRN but requires a more elaborate management for queues and disk space.

# 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 -n 1 > /var/updatelink/disablebrnntp || \
        echo -n 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 configuration 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_vlan);
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;
```

# Appendix C

# Scripts

### The SDP activation script

```
#!/bin/sh

# simple example SDP start/stop script
# $Id: activate,v 1.2 2005/11/25 03:42:11 cvs Exp $

p=/tmp/clickrun
incomingdir=/tmp/storage
WLAN=wlan0
action=$1
if test -n "$action" ; then
  shift
else
  action=help
fi
sdpdir=$1
if test -n "$sdpdir" ; then
  shift
else
  # use default, if no dir given - either flash or RAM
  sdpdir=/var/updatelink
fi

beginsig="-----BEGIN SIGNATURE-----"
hash=sha1
remount="touch /var/updatelink/ || mount -o remount,rw /dev/root /"
revision='$Revision: 1.2 $'

#eval $remount
mkdir -p $incomingdir $sdpdir/temp
```

```
cd $sdpdir
case "$action" in
  start)
    echo "starting click"
    rm -rf temp/*
    rm -rf $p
    mkdir -p $p temp library
    ver=$(cat current_id)
    cd current
    echo "starting version $ver $revision" > $p/debug.out
    echo $* >> $p/debug.out
    ls -l >> $p/debug.out
    gzip -cd click.gz > $p/click
    chmod a+x $p/click
    SSID=$(/sbin/ifconfig $WLAN | awk '/HWaddr/{print $5}' |\
      awk -F ":" '{print $4$5$6}')
    sed -e "s/##brnSSID##/brn$SSID/" config.click > $p/current.click
    cd $p
    export CLICKPATH=$sdpdir/current
    click-align current.click > current.aligned.click
    ./click current.aligned.click <&- &
#     ./click current.aligned.click <&- >/dev/null 2>&1 &
    echo $! > click.pid
    ;;
  stop)
    echo -n "stopping click"
    pid=$(cat $p/click.pid)
    (test -n "$pid" && kill $pid) || /usr/bin/killall click 2> /dev/null
    n=0;
    while [ $n -lt 70 ] && kill $pid 2>/dev/null ;
      do echo -n . ; n=$(expr $n + 1); sleep 1;
    done
    echo
    rm -f $p/click.pid
    ;;
  canceldownload)
    ver=$1
    test -n $ver || exit 1
    rm -rf $incomingdir/$ver
    ;;
  add)
    eval $remount
    newver=$1 ; shift
    mkdir -p temp library
    # asssuming the new version is not there yet
    test -n $newver || exit 1
```

```
    rm -rf library/$newver
    chmod a+x $incomingdir/$newver/activate
# this is a non-atomic operation, because it copies from ram to flash
    mv $incomingdir/$newver temp/
# this is nearly atomic
     mv temp/$newver library/
    ;;
  schedule)
    newver=$1 ; shift
    time1=100
    time2=$(expr $time1 \* 2)
    (
    while sec=$(expr $(head -2 library/$newver/meta-info|tail -1) \
        - $(date +%s)) && test "$sec" -gt $time2 ; do
      echo $sec seconds to go ;
      sleep $time1 ;
    done
    if test "$sec" -gt 0 ; then sleep $sec ; fi
    $0 switch $sdpdir $(cat current_id) $newver
    ) &
    ;;
  switch)
    eval $remount
    oldver=$1 ; shift
    newver=$1 ; shift
    echo "switching from $oldver to $newver"
    test -n "$oldver" || exit 1
    test -n "$newver" || exit 1
    test $oldver = $newver && { $0 restart ; exit 0; }
    touch $sdpdir/update_in_progress
    current/activate stop $sdpdir
    echo $newver > current_id
    rm -f current ; ln -sf library/$newver current
    rm -rf library/$oldver
    current/activate start $sdpdir
    rm -f $sdpdir/update_in_progress
    ;;
  verify)
    # check signature of a newly downloaded meta-info file
    echo $0 verifying authenticity of $1
    sed "/$beginsig/{q}" $1 > /tmp/meta-info.hash
    grep -A1000 -- "$beginsig" $1 | openssl enc -a -d |\
      openssl rsautl -verify -inkey devel.pub -pubin |\
      sed 's/$/  \/tmp\/meta-info.hash/' | ${hash}sum -c
    exit $?
    ;;
```

```
  restart)
    $0 stop
    $0 start
    ;;
  status)
    if [ -e $p/click.pid ] ; then
      if kill -0 $(cat $p/click.pid) 2>/dev/null ; then
        echo running
        exit 0
      fi
      echo dead
      exit 7
    fi
    echo "unused "
    exit 6
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|status|add|switch|schedule}"
    exit 1;
    ;;
esac
```

# List of Figures

# 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

http://pdos.csail.mit.edu/roofnet/

[Netgear] Netgear, http://www.netgear.com/

[Gentoo] Linux Documentation http://www.gentoo.org/doc/en/handbook/

[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", 1997

http://www.w3.org/TR/NOTE-OSD

[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

[Sapphire] "The Spread of the Sapphire/Slammer Worm", 2003

`http://www.cs.berkeley.edu/~nweaver/sapphire/`

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

[Zimm1980] H. Zimmermann "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communications, 1980

[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

## 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