Humboldt University Berlin
Computer Science Department
Systems Architecture Group

Rudower Chaussee 25
D-12489 Berlin-Adlershof
Germany

Phone: +49 30 2093-3400
Fax: +40 30 2093-3112
http://sar.informatik.hu-berlin.de

Softwareentwicklung für drahtlose Maschennetzwerke Fallbeispiel: BerlinRoofNet

HU Berlin Public Report
SAR-PR-2007-06

September 2007

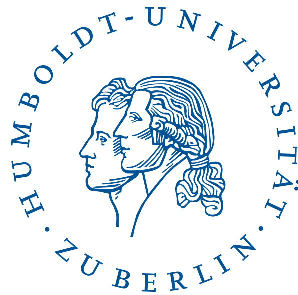
Author(s):
Mathias Jeschke

Humboldt-Universität zu Berlin
Institut für Informatik
Lehrstuhl für Systemarchitektur

Betreuer: Anatolij Zubow
Gutachter: Prof. Dr. Jens-Peter Redlich

Studienarbeit

Softwareentwicklung für drahtlose Maschennetzwerke – Fallbeispiel: BerlinRoofNet –



Mathias Jeschke

14. September 2007

Zusammenfassung

Für die Entwicklung von Softwarekomponenten für drahtlose Maschennetzwerke (Community-Netzwerke) sind geeignete Werkzeuge zur Erstellung, Deployment, Ausführung sowie Testen und Debugging unabdingbar.

Im Unterschied zu herkömmlichen Systemen handelt es sich hierbei um eingebettete Geräte mit geringen Ressourcen (CPU, RAM, Netzwerk und Sekundärspeicher), wodurch sich die Softwareentwicklung (Cross-Compilation und -Debugging) bzw. die zur Verfügung stehenden Werkzeuge (Software-Verteilung) stark unterscheiden.

Im Kontext dieser Arbeit werden Verfahren aus dem BerlinRoofNet aufgezeigt und diskutiert. Das dabei entstandene Framework kann leicht an sich ändernde Gegebenheiten (z.B. Hardware) angepasst werden.

Inhaltsverzeichnis

1 Einleitung **5**
- 1.1 Anforderungen an die Hardware . . . 6
  - 1.1.1 Plattform . . . 7
  - 1.1.2 Ausstattung . . . 7
- 1.2 Herausforderungen an die Softwareentwickler . . . 9

2 Erstellung **10**
- 2.1 Native Binärprogramme vs. Interpretierte Software . . . 10
- 2.2 Toolchain . . . 11
  - 2.2.1 OpenWrt-Toolchain erstellen . . . 11
  - 2.2.2 Eigene Programme für eine Zielplattform erstellen . . . 12
- 2.3 Bibliotheken und Schnittstellen . . . 13
  - 2.3.1 C-Bibliothek . . . 13
  - 2.3.2 Userspace-Netzwerkschnittstellen unter UNIX/Linux . . . 13
  - 2.3.3 Click-Router-API . . . 15

3 Deployment **16**
- 3.1 Begriffe . . . 16
- 3.2 Methoden . . . 17
  - 3.2.1 NFS-Server . . . 17
  - 3.2.2 ipkg-Paket . . . 17
  - 3.2.3 Firmware-Images . . . 17
  - 3.2.4 Software Distribution Platform (SDP) . . . 18
  - 3.2.5 Vergleich . . . 18
- 3.3 Beispiel: Firmware-Installation auf einem Netgear WGT634U . . . 19
  - 3.3.1 Allgemeines . . . 19
  - 3.3.2 Partitionierung eines WGT634U . . . 19
  - 3.3.3 Aufbau eines Firmware-Images . . . 20
  - 3.3.4 Austausch des originalen Systems . . . 20
  - 3.3.5 Update aus einem bestehendem System . . . 21
- 3.4 Pakete für OpenWrt-Systeme . . . 22

4 Ausführung **23**
4.1 Linux-Betriebssystem 23
4.1.1 Distributionen 23
4.1.2 Kernel 23
4.2 USB-Boot 24
4.2.1 Motivation 24
4.2.2 Einschränkungen 24
4.2.3 Variante 1: Anpassungen am Kernel 24
4.2.4 Variante 2: Anpassungen am RootFS 25
4.2.5 Vor- und Nachteile 26

5 Testen und Debugging **27**
5.1 Aspekte 27
5.2 BerlinRoofNet-Testbed 27
5.2.1 Serial-connected Node 27
5.2.2 Wired Backbone 28
5.2.3 Wireless Backbone 29
5.2.4 Outdoor Testbed 30
5.2.5 Vor- und Nachteile 30
5.3 Common System Base 31
5.3.1 Linux Diskless-Clients 31
5.3.2 NFS-Root 32
5.3.3 Homeverzeichnisse 34
5.4 Debugging 35
5.4.1 Remote-Debugging 35
5.4.2 Debugging-Beispiel 35
5.4.3 Tipps 36

6 Zusammenfassung und Ausblick **37**
6.1 Zusammenfassung 37
6.2 Ausblick 37

A USB-Boot **38**
A.1 Kernel-Variante 38
A.2 RootFS-Variante 41

B VPN-Gateway **43**

C Remote-GDB-Skript **44**

1 Einleitung

In der Informationstechnologie gibt es ein ungeschriebenes, nach dem Intel-Mitgründer Gordon Moore benanntes Gesetz: das „Mooresche Gesetz“. Es besagt, dass sich die Dichte von Transistoren auf einem Schaltkreis (also auch Prozessoren, Speicher, etc.) innerhalb von 18 Monaten (nach Moore eher alle 24 Monate) stets verdoppelt.

Durch die industrielle Massenproduktion von Computer-Komponenten herrscht in der Hardware-Branche ein Preiskampf, der es ermöglicht, dass drahtlose Zugangspunkte (ugs. WLAN-Router) heute die gleiche Rechenleistung wie gut ausgestattete Desktop-PCs vor 10 Jahren haben, dabei aber weniger als ein Zehntel kosten und zudem noch Funk-Karten nach WLAN-Standard IEEE [802.11] aufweisen.

Das *BerlinRoofNet-Projekt* [BRN] hat sich zum Ziel gesetzt, Technologien für Funknetzwerke auf Basis dieser Geräte zu entwickeln. Ein solches Netzwerk (engl. *mesh network*), wie beispielsweise in Abbildung 1.1, kann dann wiederum die Grundlage für andere Forschungszweige sein, zum Beispiel die METRIK-Projekte [MTRK], aber auch für so genannte Community-Netze, die dann ihren Nutzern zahlreiche Dienste und Ressourcen wie einen Internetzugang zur Verfügung stellen.

Ein Knoten ist, im Kontext dieser Arbeit, ein Teilnehmer im Maschennetzwerk, der primär eigene Netzwerkpakete erzeugt (z.B. Sensordaten) und sekundär fremde Pakete anderer Teilnehmer weiterleitet.

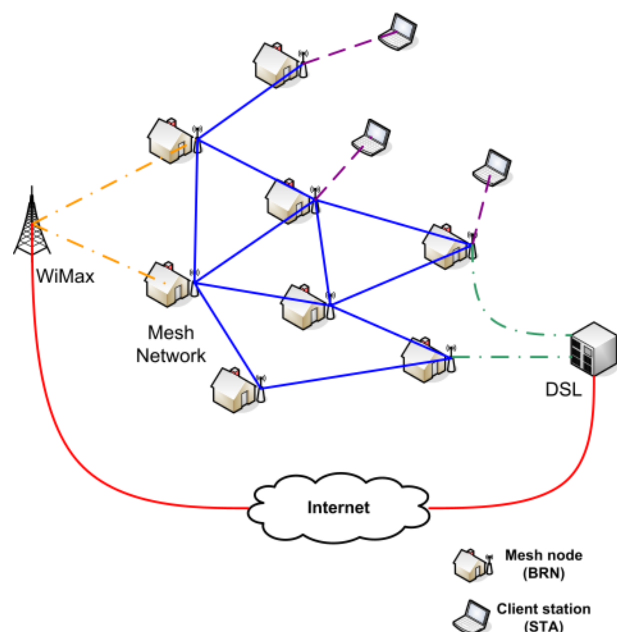


Abbildung 1.1: Typische Gestalt eines Community-Maschennetzwerkes

Das angestrebte Netzwerk soll folgende Kriterien erfüllen:

- **Maschennetzwerk** beschreibt eine Netzwerktopologie[1], in der mehrere Teilnehmer (Kno-

[1] Eine Netzwerktopologie beschreibt die physische Struktur der Verbindungen zwischen den Knoten, z.B. Bus-, Ring- oder Sterntopologie.

ten) direkt miteinander verbunden sind. Ein Knoten baut zu allen anderen in Funkreichweite befindlichen Knoten eine Verbindung auf. Im Gegensatz zu kabelgebundenen Netzwerktopologien kann sich aufgrund von Mobilität der Knoten bzw. der zeitlichen Veränderung der Funkeigenschaften des Mediums die Topologie schnell und unvorhersehbar ändern. Zwei Knoten die sich nicht in unmittelbarer Funkreichweite befinden, sind somit auf die Hilfe anderer Knoten (Relays) angewiesen. Dieses Verfahren wird Multi-Hop-Übertragung bezeichnet. Ferner ermöglicht diese Topologie durch mehrere Pfade vom Sender zum Empfänger einerseits eine bessere Ausfallsicherheit und andererseits lassen sich mehrere unabhängige Pfade zur besseren Ausnutzung der Netzkapazität einsetzen.

- **Multi-Hop-Fähigkeit** ermöglicht es, logische Verbindungen über größere Entfernungen mit Hilfe von anderen Router-Knoten des Netzes aufzubauen, die anderenfalls nicht möglich wären. Die maximale Reichweite eines so genannten *Hops* – von einem Knoten zum nächsten – beträgt etwa 50 Meter innerhalb und 300 Meter außerhalb von Gebäuden.

- **Selbstorganisation und Autokonfiguration**[2] ist die Voraussetzung für einen Betrieb ohne administrativen Eingriff eines *Operators* in die Knoten, selbst bei Änderungen der Netzwerktopologie.

- **Dezentralität** – essenzielle Dienste (z.B. DHCP[3], DNS[4]) werden nicht von speziell ausgewiesenen Knoten, sondern als *verteilte* Dienste durch alle Netzteilnehmer erbracht. Dieser Punkt ist eine direkte Folgerung aus der

- **Ad-Hoc-Fähigkeit** – es können, ohne vorherige Abstimmung mit anderen Knotenbetreibern, jederzeit Knoten dem Netz hinzugefügt und entfernt werden. Eine solche Aktion sollte die Funktionsweise des Netzes nicht beeinträchtigen.

1.1 Anforderungen an die Hardware

Aus der Zielstellung des BRN ergibt sich eine Reihe von Anforderungen an die Hardware, die im Netzwerk eingesetzt werden soll.
Soll die Technik des BRNs in der Praxis zum Aufbau von Netzwerk-Infrastrukturen dienen, so ist es für eine akzeptable Flächendeckung unumgänglich, eine gewisse kritische Menge an Knoten zu erreichen. Der Idealfall wäre sicherlich, wenn viele Breitband-Internet-Teilnehmer die notwendige Software auf ihre meist schon vorhandenen WLAN-Router aufspielen könnten.

Als Teil der Netzwerk-Infrastruktur sollte ein Knoten zudem praktisch rund um die Uhr online sein, was eine Forderung nach geringem Stromverbrauch nach sich zieht. Momentan im Einsatz

[2] Netzwerkparameter (IP, DNS, etc.) können ohne Eingriff des Endnutzers selbständig ermittelt und eingerichtet werden.
[3] Dynamic Host Configuration Protocol – Protokoll zur IP-Autokonfiguration [Dro97]
[4] Domain Name System – Hierarchisch aufgebauter Namensdienst für IP-Netzwerke [Moc87a], [Moc87b]

befindliche Geräte benötigen inklusive Netzteil weniger als 10 Watt – in der Summe ca. 80 kWh pro Jahr.

1.1.1 Plattform

Auf dem Markt der Embedded Systems (dt. Eingebettete Geräte) sind bislang im Gegensatz zum Desktop-Segment noch keine x86-kompatiblen Systeme verbreitet.

Da für die BerlinRoofNet-Knoten Linux [Lin] als Betriebssystem zu Einsatz kommt, muss dieses für jede unterstützte Zielplattform verfügbar sein. Dies erfordert eine Unterstützung der Plattform von der *GNU Compiler Collection* (GCC). Bei Verwendung der (flexibel einsetzbaren) Atheros-WLAN-Karten ist darauf zu achten, dass ein *Hardware Abstraction Layer* (HAL) für die Architektur vorhanden ist. Dieser vom Hersteller zur Verfügung gestellte Software-HAL soll sicherstellen, dass die Karten nur innerhalb erlaubter physischer Parameter (z.B. Sendestärke) betrieben werden. Er ist deshalb nur als (plattformabhängiger) Binärcode für folgende relevante Architekturen verfügbar [MAD]:

- x86,
- x86_64,
- MIPS,
- ARM,
- Sparc,
- PowerPC.

Wünschenswert, aber nicht notwendig, ist eine OpenWrt-Unterstützung[5]. Dadurch steht eine Vielzahl an Softwarepaketen zur Auswahl[6], die meist ohne Anpassung auch auf „neuen“ Plattformen sofort einsetzbar ist.

1.1.2 Ausstattung

Es ist offensichtlich, dass jeder BRN-kompatible Router zunächst mindestens einen WLAN-Adapter benötigt. Ferner muss dieser Adapter aufgrund der eingesetzten Click-Software (siehe Kapitel 2) das Übertragen (*Injection*) von *Wifi-Frames* erlauben. Das heißt, es soll möglich sein, eigene komplett in Software generierte Frames nach IEEE 802.11 zu senden und auch die Antworten mit der gleichen Software zu verarbeiten.

Viele der eingesetzten Geräte haben nur wenig Sekundärspeicher in Form von Flash-Bausteinen, die in der Regel nicht ausgetauscht werden können. Daher ist das Vorhandensein einer

[5] OpenWrt ist eine Linux-Distribution für Embedded-Systeme. OpenWrt unterstützt zahlreiche Hardwareplattformen und bietet eine umfangreiche Sammlung von Anwendungsprogrammen an [WRTa].

[6] Einen Überblick gibt das OpenWrt-Entwickler-Repository [WRTb]

Schnittstelle zum Anstecken eines USB-Sticks, einer Compact-Flash-Karte o.ä. für manche Anwendungsfälle (z.B. das Mono-Projekt [MONOa], [MONOb] und andere Projekte, die große Programmbibliotheken mitbringen) zwingend notwendig.

Im BRN wurden bislang folgende Hardware-Plattformen eingesetzt:

- Linksys WRT54GS,
- Netgear WGT634U,
- Embedded-x86-Plattformen (WRAP, Soekris, Routerboard),
- verschiedene x86-kompatible Systeme (PCs).

Linksys WRT54GS

Der populäre WRT54GS[7] von Linksys war der erste im Projekt genutzte WLAN-Router. Er verfügt über eine MIPS32-CPU (Little Endian) von Broadcom mit 200 MHz, 32 MB RAM, 8 MB Flash, einen 5-Port-Ethernet-Switch (mit VLAN-Funktionalität) und einen Broadcom-WLAN-Chip. Wegen des unflexiblen (closed-source) WLAN-Treibers konnte der Linksys nur eingeschränkt mit der Click-Software zusammen eingesetzt werden. Weiterhin können die Geräte als flexible Ethernet-Switches und WLAN-Clients verwendet werden.

Netgear WGT634U

Als Nachfolger für *den* BRN-Knoten wurde der WGT634U von Netgear auserkoren. Die Hardware dieser Router ist relativ ähnlich zum WRT54GS, da er ebenfalls ein integriertes Mainboard der Firma Broadcom benutzt.

Vorteile des WGT634U sind vor allem die **USB-Schnittstelle** für zusätzliche Peripherie-Geräte (Sekundärspeicher, GPS-Empfänger für Geo-Routing, Webcams und vieles mehr), der **Atheros-WLAN**-Adapter und ein flexiblerer Bootloader (mit Unterstützung für Netzwerk-Boot, siehe Kapitel 5.3.2). Der WLAN-Adapter ist außerdem über einen **MiniPCI**-Anschluss angeschlossen und kann gegen andere MiniPCI-Adapter (z.B. 900 MHz-Funknetzwerkkarten) ausgetauscht werden, sofern die nötigen Treiber verfügbar sind. Für Rettungsversuche steht zudem ein Anschluss für die serielle Konsole bei allen Geräten zur Verfügung (siehe Kapitel 5.2.1).

Embedded-x86-Plattformen

Parallel zum Netgear-Router kommen vereinzelt auch Systeme mit x86-kompatibler CPU zum Einsatz, die allerdings deutlich teurer, dafür aber noch flexibler konfigurierbar sind. Getestet wur-

[7] Die Popularität ist darin begründet, dass er der erste kostengünstige (≈ 80 €) WLAN-Router mit installiertem Linux-Betriebssystem war. Außerdem hat der WRT54GS eine im Vergleich zu herkömmlichen PCs geringe Stromaufnahme.

den bisher das *WRAP-Board* der Firma PC Engines und von Soekris der *net4826*. Beide Geräte verwenden das Embedded-x86-Design vom Typ *AMD Geode SC1100* und sind aus Softwaresicht fast identisch. Als Vorteil der Boards sind die vorhandenen MiniPCI-Slots und der selbstbestückbare Compact-Flash-Slot zu nennen, die Konfigurationen mit 2 WLAN-Adaptern und ausreichend Sekundärspeicher ermöglichen.

1.2 Herausforderungen an die Softwareentwickler

Die eingesetzten BRN-Knoten besitzen keine einheitliche Hardwareplattform, die neben dem Befehlssatz auch teilweise andere Daten-Repräsentationen (Endianess) aufweisen.

Als eingebettete Geräte haben die Knoten geringere Ressourcen (CPU, RAM, Netzwerk, Sekundärspeicher) im Vergleich zu herkömmlichen Desktop-PCs. Dieses Manko zeigt sich beispielsweise deutlich beim Debugging von Software auf den Knoten selbst und bei der Ausführung anderer speicherintensiver Programme. Folglich werden auf den BRN-Knoten stark modifizierte Programme, aber auch Systembibliotheken, z.B. aus dem uClibc- und Busybox-Projekt, benutzt (siehe Kapitel 2.3.1). Dies muss beim Linken von Entwicklersoftware gegen diese Bibliotheken nicht immer zur gleichen Syntax und Semantik führen.

Für Entwickler, die nur über die Hardware selbst und nicht über Entwicklerkits o.ä. verfügen, können fehlende Dokumentation der Hardware oder nicht funktionierende Toolchains (siehe Kapitel 2.2) Probleme bereiten.

Die Softwareverteilung (engl. *Deployment*) und -aktualisierung im BRN gestaltet sich aufgrund der unterschiedlichen Hardware, aber auch durch die geographische Verbreitung der Knoten als nicht-triviale Aufgabe.

2 Erstellung

Aus der Vielfalt der Hardwareplattformen (siehe Kapitel 1) lässt sich erahnen, dass die Entwicklung und Verteilung von Software für das BRN sich etwas anders gestaltet als bei PC-Systemen. Dies liegt zum einen daran, dass ein breites Spektrum unterschiedlicher Hardware eingesetzt wird, und zum anderen, dass viele BRN-Knoten nur mit geringen Ressourcen (z.B. CPU und Speicher) ausgestattet sind.

Zur Unterstützung der Software-Entwicklung existieren für diese Plattformen Compiler und andere wichtige Werkzeuge. Diese werden unter dem Begriff **Toolchain** zusammengefasst und sollen Thema des zweiten Abschnittes sein.

Bei der Softwareentwicklung praktisch unverzichtbar sind Programmbibliotheken, sei es, um die vorhandene Hardware bequem anzusprechen oder für allgemeine Hilfsfunktionen (z.B. Sortieren, Suchen oder Speicherverwaltung). Im dritten Teil dieses Kapitels werden die häufig verwendeten und für das BRN relevanten Bibliotheken besprochen.

2.1 Native Binärprogramme vs. Interpretierte Software

Anwendungsentwickler stehen oft vor der Wahl, ob sie ihre Software in einer Programmiersprache codieren, die übersetzt (kompiliert) und dann als plattformabhängiges Binärprogramm ausgeführt wird, oder ob sie eine interpretierte Sprache einsetzen [ASU86]. Interpretierte Programme werden teilweise auch als Skripte bezeichnet. Daneben existieren auch Mischformen: Java-Programme werden beispielsweise zuerst für eine hypothetische Maschine in den so genannten Bytecode übersetzt, der anschließend von einer *Java Virtual Machine* (JVM) interpretiert wird [Ull06].

Beide Varianten bieten Vor- und Nachteile. Native Programme werden in der Regel schneller ausgeführt und haben weniger Speicherbedarf, da kein Interpreter geladen werden muss. Allerdings muss für jede Hardwareplattform eine eigene Variante des Binärprogramms vorhanden sein, was zusätzliche Anforderungen an die Softwareverteilung stellt. Skripte sind leicht anpassbar und zu verteilen – oft durch einfaches Kopieren. Leider lassen sich nicht alle Softwareprojekte mit Skripten realisieren, weil die notwendige Ausführungsumgebung fehlt. Python- oder Java-Programme laufen zum Beispiel (bisher) nicht im Linux-Kernel. Deshalb werden auf BRN-Knoten beide Software-Formen zu finden sein, Skripte meist für die Steuerung von Betriebssystem-Komponenten (Init-Skripte) und daneben viele Binärprogramme. Die Click-Software (siehe Abschnitt 2.3.3) ist ebenfalls ein plattformabhängiges Binärprogramm, das auf BRN-Knoten anzutreffen ist

und so genannte Click-Skripte interpretiert. Zurzeit wird an Möglichkeiten geforscht, auch andere Umgebungen wie Java oder Mono [MONOa] für alle BRN-Plattformen verfügbar zu machen.

2.2 Toolchain

Eine Toolchain beinhaltet alle Werkzeuge, um Software aus dem Quelltext in ihre binäre Entsprechung einer Hardwareplattform zu überführen. Darunter zählen meist (Cross-)Compiler, Assembler, Linker, aber auch Werkzeuge, um ein Firmware-Image zu erstellen, oder ein Debugger. Der Begriff der „Kette“ (*engl. chain*) leitet sich aus der Verwendung der Werkzeuge ab. Diese werden üblicherweise hintereinander „in einer Kette“ genutzt, um das Gesamtprodukt (hier: Binärprogramm) zu erhalten.

Im Normalfall möchte man den Compiler, der viele Ressourcen für den Übersetzungsprozess benötigt, nicht auf der Zielplattform, sondern auf dem eigenen, leistungsstarken Entwicklungsrechner ausführen. In diesem Fall spricht man von einem **Cross-Compiler**, welcher Maschinencode für eine andere Plattform erzeugt.

Im BRN wird für diesen Entwicklungsschritt heute die Toolchain des OpenWrt-Projekts eingesetzt. An dieser Toolchain vorteilhaft ist, dass sie:

- für viele Hardwareplattformen verfügbar ist,
- herstellerunabhängig ist, da sie als Open-Source verfügbar ist,
- Build-Skripte (Makefiles und Patches) für viele Softwarepakete bereits enthält,
- leicht erweiterbar ist.

Bevor das für die meisten BRN-Entwickler notwendige Erzeugen einer eigenen Toolchain erklärt wird, sei darauf hingewiesen, dass dies für Anwendungsentwickler in der Regel nicht nötig ist. Hierfür bietet das OpenWrt-Projekt für jede Plattform ein Entwicklerset (*OpenWrt-SDK* genannt) an[1]. Dieses enthält die notwendigen Werkzeuge in Form eines komprimierten tar-Archives, mit dem sich nach dem Auspacken sofort die eigenen C- und C++-Programme übersetzen lassen.

2.2.1 OpenWrt-Toolchain erstellen

Um eine OpenWrt-Toolchain zu erstellen, die für die BRN-Software empfohlen wird, benötigt man zunächst den Quelltext, der zum einen aus einem tar-Archiv von der Homepage[2] oder direkt aus dem Entwickler-Repository[3] heruntergeladen werden kann. Hier sollte man nach Möglichkeit auf veröffentlichte Versionen (*Releases*) setzten, um die Interoperabilität mit anderen Entwicklern zu erhöhen. Die aktuelle und in dieser Arbeit verwendete Version ist *Kamikaze 7.07*.

[1] Zum Entwickeln vom Kernelmodulen wird in jedem Fall eine „komplette“ Toolchain inklusive Kernelquellen benötigt.

[2] http://downloads.openwrt.org/kamikaze/7.07/kamikaze_7.07.tar.bz2

[3] `svn co https://svn.openwrt.org/openwrt/tags/kamikaze_7.07`

Nach dem Auspacken bzw. Auschecken auf dem Entwicklungsrechner und sofern auf diesem alle notwenigen Software-Pakete installiert sind, (C-Compiler, Make, Autoconf, Bison, Flex, zlib-devel, python-devel, ...) kann man auswählen, für welche Hardwareplattform (und welches Modell) die Toolchain erzeugt werden soll:

```
$ cd kamikaze_7.07
$ make menuconfig
```

Die Menüsteuerung ist an der des Linux-Kernels angelehnt und sollte selbsterklärend sein. Nach dem Beenden des Setups schreibt `make` die Konfiguration in die Datei `.config`, welche Grundlage des späteren Übersetzungsprozesses ist.

Im Anschluss kann man den Build-Prozess starten:

```
$ make .config          # nur "make" baut alles (Toolchain, Kernel, Programme)
$ make tools/install
$ make toolchain/install
```

Nun sollte die Toolchain einsatzbereit sein. Als Indiz kann man auf die Existenz des Cross-Compilers unter `staging_dir_<arch>/bin/<arch>-linux-gcc` prüfen[4].

2.2.2 Eigene Programme für eine Zielplattform erstellen

Hello World-Beispiel

Ob der Compiler korrekt funktioniert, kann man durch ein kleines Testprogramm überprüfen. Hierfür kann ein solches Hello-World-Programm in C verwendet werden:

```
#include <stdio.h>

int main(void) {

    printf("hello, world\n");
    return 0;
}
```

Zum Übersetzen benutzt man den (Cross-)Compiler, wie folgt: (am Beispiel „mipsel“)

```
$ staging_dir_mipsel/bin/mipsel-linux-gcc -static -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable, MIPS, version 1 (SYSV), statically linked, not
    stripped
```

Das Programm sollte statisch gelinkt werden, um (für einen ersten Test) unabhängiger von der Systeminstallation auf der Zielplattform zu sein.[5]. Das Programm „`hello`“ sollte das erwartete Ergebnis auf der Zielplattform (mipsel) liefern.

[4] `<arch>` steht hierbei für die Architektur, z.B. „`mipsel`“ oder „`i386`“

[5] So wird nur ein funktionierender Linux-Kernel benötigt.

2.3 Bibliotheken und Schnittstellen

2.3.1 C-Bibliothek

Die C-Bibliothek ist *die* Bibliothek, ohne die sich praktisch (fast) kein UNIX- oder Linux-Programm überhaupt übersetzen lässt, das nativ auf der Maschine laufen soll. PC-Linux-Systeme nutzen durchgängig die GNU-C-Bibliothek (glibc) [FSFb], Embedded-Systeme wie im BRN jedoch eine abgespeckte C-Bibliothek, z.B. die uClibc [uClb].

Bei der Portierung und Entwicklung von Software kann dies zu Problemen führen, wenn Funktionen der glibc genutzt werden, die in der C-Bibliothek des Zielsystems nicht vorhanden sind oder eine andere Semantik haben. Deshalb sind zum Teil Patches nötig, die „glibc-Software" „uClibc-tauglich" machen.

2.3.2 Userspace-Netzwerkschnittstellen unter UNIX/Linux

Sockets

Die für die Netzwerkkommunikation bekannteste und wohl am meisten genutzte Schnittstelle sind sicherlich die BSD-Sockets, die auch auf anderen Nicht-UNIX-Betriebssystemen (z.B. Microsoft Windows) zu finden sind.

Sockets erlauben es, (unprivilegierten) *Anwendungsprogrammen*[6] untereinander Daten auszutauschen. Dies kann über TCP und UDP auf der Basis des Internet Protokolls (IP) oder UNIX-Domain-Sockets geschehen. Ein Socket – eine logische Verbindung – wird im Wesentlichen durch 4 Parameter beschrieben:

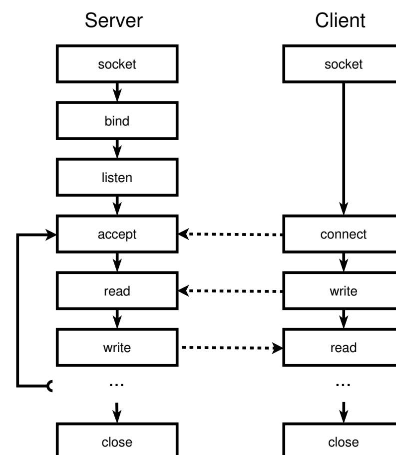


Abbildung 2.1: Syscalls für eine TCP-Verbindung

- der IP-Adresse des Absenders,
- dem TCP- bzw. UDP-Port des Absenders,
- der IP-Adresse des Empfängers,
- dem TCP- bzw. UDP-Port des Empfängers.

Eine solche Verbindung muss durch den Aufruf des `socket`-Syscalls auf beiden Seiten vorbereitet werden. Im Anschluss ruft bei TCP-Verbindungen eine Seite (üblicherweise der *Server* im

[6] *Unprivilegiert* meint in diesem Kontext, dass solche Programme im System nur mit eingeschränkten Rechten laufen.

Client-Server-Modell) die Syscalls `bind`[7], `listen` und `accept`, die andere Seite (der *Client*) `connect` auf. TCP garantiert dann die verlustlose Zustellung und korrekte Reihenfolge der empfangenen Daten. Ein TCP-Socket ist daher mit einer bidirektionalen Pipe zwischen zwei entfernten Prozessen vergleichbar. Abbildung 2.1 zeigt den Zusammenhang schematisch.

Für UDP-Verbindungen unterscheidet sich das Vorgehen im Vergleich zu TCP dahingehend, dass die Rollen des Servers und des Clients nicht durch Transportschicht, sondern vielmehr durch die Anwendungsschicht festgelegt werden, sofern eine solche Rollenverteilung überhaupt gewollt ist[8]. Daher gibt es für UDP neben `bind` auch nur zwei Syscalls: `sendto` und `recvfrom`, die, wie die Namen vermuten lassen, für das Senden und Empfangen von Daten-Paketen (*Datagramme*) zuständig sind.

Die Popularität von Sockets vor allem zusammen mit TCP zeigt, dass diese ein recht großes Spektrum der Interprozess-Kommunikation abdecken. Eine wichtige Beschränkung ist jedoch, dass diese Form der Sockets nur TCP und UDP aber keine anderen auf IP aufsetzende Protokolle (z.B. ICMP) oder sogar andere Layer-3-Protokolle wie ARP oder proprietäre Protokolle (z.B. das BRN-Protokoll) unterstützen.

RAW-Sockets

Zur Kommunikation mit Protokollen, die auf IP aufbauen (z.B. ICMP, IGMP oder OSPF), kann man so genannte RAW-Sockets verwenden, was unter UNIX aber nur dem root-Nutzer gestattet ist. Eine populäre Anwendung von RAW-Sockets ist das Programm „ping“.

Packet-Sockets

Benötigt ein Userspace-Programm Zugriff auf vollständige Link-Layer-Pakete, kann hierzu unter Linux ein Packet-Socket eingesetzt werden.

libpcap

Bei der Programmierung von Netzwerk-Analyse-Werkzeugen unter UNIX besteht das Problem, dass sich die Link-Layer-Schnittstellen zwischen UNIX-Derivaten unterscheiden. Die pcap-Bibliothek [Pcap] bietet eine entsprechende Abstraktion zum Abgreifen (*dump*) und Einspielen (*inject*) derartiger Pakete. Die Bibliothek wird zum Beispiel im Netzwerksniffer
`tcpdump`, aber auch in der Click-Software benutzt.

[7]Der Syscall `bind` reserviert beim Systemkern eine gewünschte Portnummer, die die Gegenseite (der Client) dann kontaktieren kann. `bind` kann auch auf der Client-Seite benutzt werden, um den „Client-Port“ auszuwählen – ohne den Aufruf legt der Systemkern die Portnummer, mehr oder weniger zufällig fest.

[8]Man betrachte z.B. Peer-to-Peer-Anwendungen

TUN/TAP

In manchen Anwendungen besteht der Wunsch, Netzwerkpakete direkt im Userspace zu bearbeiten und im Anschluss die modifizierten Pakete mit Hilfe des Betriebssystems und dessen Netzwerk-Stack entweder an lokale Prozesse oder per Routing über das Netzwerk zu anderen Rechnern zu übertragen. Viele UNIX-Systeme und auch Windows[9] bieten für diesen Zweck einen virtuellen Netzwerkadapter namens TUN/TAP an [Kra02]. Dieser Treiber legt dabei im Betriebssystem einerseits ein „virtuelles“ Netzwerkgerät mit dem Namen tun*N* bzw. tap*N* an (wobei N von 0 an hochgezählt wird; je nach Anzahl der verknüpften Prozesse) und andererseits wird eine zeichenorientierte Gerätedatei (engl. *character device*) `/dev/tun` bzw. `/dev/net/tun`[10] eingerichtet.

Schreibt nun ein Userspace-Prozess ein Netzwerkpaket in diese Gerätedatei, sorgt der TUN/-TAP-Treiber dafür, dass dieses Paket aus dem Netzgerät im Kernel „herausfällt“ – genau so wie es der Treiber für einen realen Netzwerkadapter realisiert. In der entgegengesetzten Richtung werden alle Pakete, die an das tun*N*/tap*N*-Gerät über den Kernel-Netzwerk-Stack „gesendet“ werden, dem Prozess als zu lesende Daten in der o.g. Gerätedatei bereitgestellt.

TUN- und TAP-Geräte unterscheiden sich in der Zuordnung der Schichten des ISO/OSI-Modells. TUN transportiert ausschließlich IP-Pakete (im Point-to-Point-Modus, OSI-Layer 3), während TAP mit Ethernet-Frames (OSI-Layer 2) arbeitet.

Der Vorteil von TUN/TAP liegt auf der Hand: Durch die Benutzung dieser wohldefinierten Kernelschnittstelle können bereits existierende (und meist ausgiebig getestete) Algorithmen des Kernels über eine etablierte Benutzersicht verwendet werden. Beispiele hierfür sind das Routing zwischen Netzwerkgeräten an Hand einer zentral verwalteten Routing-Tabelle[11], aber auch Mechanismen wie Paketfilter und Masquerading bzw. Network Address Translation (NAT). So können Filter realisiert werden, die Ethernet-Frames transparent (im Userspace) modifizieren.

2.3.3 Click-Router-API

Eine weitere API wird auf BRN-Knoten durch die Click-Software [KMC+00] definiert. Dieser um eigene Module (*Elemente* genannt) erweiterbare Interpreter liest die eingegebenen Skripte ein und manipuliert den Paketfluss im Kernel entsprechend.

Als Einstieg in diese Sprache eignen sich die Beispielskripte [CLKa] und das Tutorial [CLKb].

[9] Per Drittanbieter, z.B. vom OpenVPN-Projekt.
[10] Der Name ist dabei abhängig von der konkreten UNIX-Distribution und wird Kernel-intern über so genannte *Major- und Minor-Nummern* addressiert.
[11] Unter UNIX und Windows mit dem Kommando `route`.

3 Deployment

Nachdem die BRN-Entwickler ihre Software mit den Werkzeugen aus Kapitel 2 kodiert und kompiliert haben, bleibt eine weitere Herausforderung: die Verteilung (engl. *Deployment*) auf die Vielzahl der BRN-Knoten, die die Software später ausführen sollen.

3.1 Begriffe

Um die im Folgenden beschriebenen Methoden des Software-Deployments in drahtlosen Maschennetzwerken besser bewerten zu können, werden einige Begriffe benötigt:

- **Versionierung** bezeichnet die *Zuordnung* von Dateien bzw. Dateisammlungen (Paketen) zu Elementen einer geordneten Menge. Als Menge (für die einzelnen *Versionen*) kommen meist Natürliche oder Rationale Zahlen, teilweise um das lateinische Alphabet erweitert, zum Einsatz. Die Versionierung ermöglicht es, die Entwicklungsgeschichte von Dateien und Programmen zu verfolgen und einzelne Versionen zu identifizieren.
- Eine **Paketverwaltung** ermöglicht das Hinzufügen, Löschen und Aktualisieren von Software (den Paketen) auf einem Rechnersystem. Die dafür benutzte Verwaltungssoftware soll außerdem eine Konsistenz-Eigenschaft[1] sicherstellen, indem eine installierte Datei nur zu *einem* Paket gehören darf[2]. Neben Dateien (Binärcode, Konfiguration, Dokumentation, etc.) beinhaltet ein Paket in der Regel noch Metainformationen. Diese werden von der Verwaltungssoftware, beispielsweise zur Erkennung von Abhängigkeiten, für die Software-Aktualisierung (engl. *Updates*) und zur Anzeige von Nutzerinformationen im Frontend benutzt.
- **Online/Offline** Lassen sich Softwarekomponenten direkt aus dem laufenden System heraus über eine Netzwerkverbindung installieren bzw. aktualisieren (*Online*) oder ist hierfür der Umweg über einen Client notwendig (*Offline*)?

[1]Falls das System vor einer Aktion als stabil anzusehen ist, soll dies auch noch nach der Aktion der Fall sein.

[2]Das Problem der so genannten „DLL-Hölle“ (engl. *DLL Hell*) bei Systemen unter Microsoft Windows soll so vermieden werden.

3.2 Methoden

Für das Deployment haben BRN-Entwickler eine Reihe von Möglichkeiten zur Auswahl, die sich aber in der Wartbarkeit und Verteilungsgeschwindigkeit[3] stark unterscheiden.

3.2.1 NFS-Server

Eine einfache Art der Softwareverteilung besteht darin, die ausführbaren Binärprogramme und notwendige Bibliotheken auf einen (zentralen) NFS-Server zu kopieren und die Programme im Anschluss von dort direkt auszuführen. Der Vorteil liegt in der primären Zeitersparnis durch den nicht vorhandenen Verwaltungsaufwand, der bei anderen Methoden, z.B. für das Bauen von Paketen, benötigt wird. Nachteile sind die fehlende Versionierung und die Notwendigkeit, einen zentralen NFS-Server zu betreiben, was die geforderten Ziele des BRN-Projekts (siehe Kapitel 1) verletzt. Bei der Software-Entwicklung, insbesondere beim Testen, kann der Einsatz dieser Methode aber durchaus nützlich sein.

3.2.2 ipkg-Paket

Das *Itsy Package Management System* [Ipkg] ist eine an das Debian Package Management (dpkg) angelehnte und für eingebettete Systeme optimierte Paketverwaltung. Sie wird in der OpenWrt-Distribution zur (nachträglichen) Installation und Aktualisierung von Softwarepaketen genutzt. Eine Versionierung ist integraler Bestandteil dieser Methode. Bei einer großen Zahl von BRN-Knoten ist dafür entweder ein zentraler Server (für die Datenablage und Steuerung) oder aber ein manueller Eingriff eines Operators notwendig. Die Erstellung eigener ipk-Pakete wird in Kapitel 3.4 erläutert.

3.2.3 Firmware-Images

Ein neuer BRN-Knoten wird in der Regel entweder mit keinem oder nur mit einem ungeeigneten (vom Hersteller unterstützten) Betriebssystem ausgeliefert. Für die Basisinstallation des Knotens muss deshalb zunächst ein Firmware-Image, das den Kernel und grundlegende Programme enthält, installiert werden – vergleichbar mit einem Abzug von Festplattendaten eines PC-Betriebssystems.

Die Möglichkeit vorgefertigte, unter Umständen komplexe Konfigurationen ausliefern zu können, hat für den Betreiber eines BRN-Knotens einen wichtigen Vorteil: Er kann den Knoten einfach einrichten und benötigt keine weiteren Kenntnisse über die einzelnen Softwarekomponenten und deren Zusammenspiel.

Nachteilig ist, dass bei einem Update (durch Austausch) Einstellungen meist „von Hand“ gesichert und wieder eingespielt werden müssen. Zudem ist der gesamte Prozess des Firmwaretau-

[3] Die Zeit von der Erstellung bis zur frühstmöglichen Ausführung von Softwarepaketen.

sches riskant, da im Falle von Fehlersituationen (z.B. Stromausfall, Softwarefehler) kein Rollback[4] möglich ist.

Die Hinzunahme neuer Hardwareplattformen wird durch die unterschiedlichen Verfahren des Firmwaretausches erschwert, da hier im Gegensatz zur Paketverwaltung keine einheitliche Schnittstelle zur Verfügung steht.

In der Regel wird bei der Veröffentlichung von Firmware-Images eine Versionierung eingesetzt. Diese wird aber normalerweise beim Einspielen eines solchen nicht dahingehend geprüft, dass nur „neuere“ Images eingespielt werden dürfen.

3.2.4 Software Distribution Platform (SDP)

Die Software Distribution Platform [Wie06] ist ein Werkzeug, dass es ermöglicht, Software in einem drahtlosen Maschennetzwerk über die Funkschnittstelle zu verteilen. Weitere Merkmale sind:

- Virale Verbreitung,
- Dateiübertragung ausschließlich über lokale Netzwerklinks (zwischen Nachbarknoten) – somit ist kein Routing notwendig,
- Automatische Benachrichtigung der Nachbarknoten bei vorhandenen Updates,
- Zeitsynchronisierung der Knoten und planbare Aktivierung (Umschalten von einer Version auf eine andere),
- Metadaten der Software können vom Entwickler signiert werden,
- Versionierung.

3.2.5 Vergleich

Die nachstehende Tabelle soll eine Zusammenfassung der angeführten Methoden liefern:

Methode	versioniert	paketbasierend	online	automatisch[5]
NFS	–	–	–	x
ipkg	x	x	x	–
SDP	x	–	–	x
Firmware	x	–	x[6]	–

[4] Ein Rollback bezeichnet das Zurückkehren auf einen (letzten) stabilen Zustand.
[5] Die Verteilungsmethode sorgt automatisch dafür, dass Aktualisierungen ohne Operator-Eingriff installiert werden.
[6] Firmwareabhängig

3.3 Beispiel: Firmware-Installation auf einem Netgear WGT634U

Wie in Abschnitt 3.2.3 erwähnt, ist die Installation eines Firmware-Images stark von der benutzten Hardwareplattform abhängig. Die Probleme bei der Generierung eines Firmware-Images sollen im Folgenden exemplarisch am Netgear WGT634U gezeigt werden.

3.3.1 Allgemeines

Ein WGT634U wird bereits von Netgear mit einem Linux-Router-System (basierend auf der Distribution des LEAF-Projekts [LEAF]) ausgeliefert.
Das System ist komplett und ausschließlich über eine Weboberfläche konfigurierbar. Zur Aktualisierung der originalen System-Software bietet diese Oberfläche dem Nutzer an, ein neues Firmware-Image hochzuladen und zu installieren. Hierbei wird der Linux-Kernel und das Root-Dateisystem komplett ausgetauscht, die Einstellungen sind davon nicht betroffen; sie sind in einer eigenen Partition abgelegt.

3.3.2 Partitionierung eines WGT634U

Der Flash-Speicher wird vom Bootloader des WGT634U logisch in 4 Partitionen gegliedert (siehe Abbildung 3.1):

1. Die boot-Partition beinhaltet den Programmcode des Bootloaders (CFE), der direkt nach dem Einschalten geladen wird.
2. Die config-Partition wird von der Netgear-Firmware zur Speicherung von Konfigurationsparametern genutzt.
3. Die os-Partition enthält den Kernel und das Root-Dateisystem. Diese Partition wird später durch den Linux-Kernel in 2 weitere Partitionen (`kernel` und `rootfs`) aufgeteilt.
4. In der nvram-Partition werden Einstellungen den Bootloader betreffend abgelegt. Zum Beispiel: *Welcher Kernel soll wie geladen werden?* (vom Flash, via TFTP, gz-komprimiert).

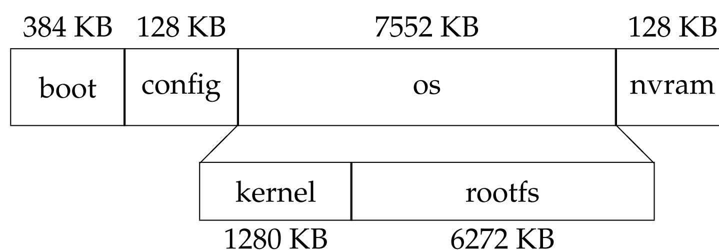


Abbildung 3.1: Partitionierung eines WGT634U

3.3.3 Aufbau eines Firmware-Images

Achtung: Die folgenden Informationen wurden (mangels Herstellerdokumentation) durch Reverse-Engineering gewonnen und können daher unvollständig sein.

Möchte ein Entwickler sein eigenes Firmware-Image erstellen, das sich über die Weboberfläche installieren lässt, muss er sich an das Format in Abbildung 3.2 halten.

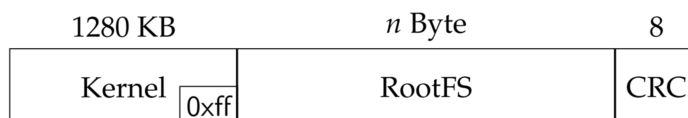


Abbildung 3.2: Format eines Firmware-Images für den WGT634U

Der Kernel muss im ELF-Format vorliegen und ausführbar sein, auch wenn der Bootloader prinzipiell gzip-komprimierte Kernel laden kann, ist dies leider nicht voreingestellt. Um dennoch komprimierte Kernel verwenden zu können, ist ein Entpackprogramm in Maschinencode notwendig, das vor das eigentliche (komprimierte) Kernelimage voranzustellen ist. Auf diese Weise lassen sich ungefähr zwei Drittel des Kernel-Codes einsparen. Die restlichen Bytes zu 1280 Kilobyte müssen mit 0xff-Bytes auffüllt werden, weil zum einen die Installationsroutine der Weboberfläche beim „Upgrade“ ein solches Feld (von 100 0xff-Bytes ab Position 1310621 der Kernel-Image-Partition) erwartet und zum anderen solche Bytes nicht auf den Flash geschrieben werden brauchen[7].

Das Format des Root-Dateisystems ist dem Entwickler relativ frei gestellt – es muss nur von „seinem“ Kernel erkannt werden und in die Flash-Partition passen. In der Praxis werden häufig die komprimierten Dateisysteme jffs2 [JFFS2] und squashfs [Sqshfs] eingesetzt.

Die, bei der angegebenen Partitionierung, maximale Größe des Root-Dateisystems beträgt 6272 KB.

3.3.4 Austausch des originalen Systems

Um die beschriebenen Unzulänglichkeiten zu umgehen und den Flash-Speicher noch effizienter auszunutzen[8], muss man eine Änderung am originalen Netgear-System vornehmen.

Die Web-Oberfläche ermöglicht es, eine (zuvor gespeicherte) Konfigurationsdatei über die Web-Oberfläche einzuspielen. Diese Funktion ist ursprünglich für das Wiederherstellen von Router-Einstellungen (Netzwerk, Dialup, WLAN, etc.) vorgesehen. Genauer betrachtet, sind diese Konfigurationsdateien nichts anderes als ein Abbild der „config“-Partition im Flash-Speicher des Routers, die ein Minix-Dateisystem [TW06] beinhaltet. Beim Zurückspielen der Einstellungen wird

[7] 0xff-Bytes sind „leere Bytes“ im Flash-Chip.

[8] Beispielsweise nutzt ein OpenWrt-Image den „freien“ Platz zwischen dem Ende des Kernel-Images und der genannten 1280kB-Grenze für das RootFS.

128 KB	100	8
MinixFS-Image (mit `config.lrp`)	Info	CRC

Abbildung 3.3: Aufbau einer Konfigurationsdatei des originalen WGT634U-Systems

die Datei bis auf Meta-Informationen 1:1 auf den Flash (in die besagte Partition) geschrieben. Eine korrekte Konfigurationsdatei ist wie in Abbildung 3.3 aufgebaut.

Das Minix-Dateisystem wiederum beinhaltet die Datei `config.lrp` (andere Dateien im Dateisystem werden ignoriert), die beim Hochfahren des Systems nach allen anderen lrp-Dateien[9] im RootFS entpackt wird. Normalerweise enthält das Archiv einen Abzug von `/etc`, aber auch andere Dateipfade werden akzeptiert.

OpenWrt-Hack-File

Für die Installation auf einem WGT634U hat das OpenWrt-Projekt für ihre Distribution eine Datei[10] veröffentlicht, welche die Beschränkungen etwas „radikal" deaktiviert. So wird beim Entpacken der OpenWrt-Einstellungsdatei während des Netgear-Systemstarts einerseits ein neuer Menüeintrag angelegt und andererseits die beiden Sicherheitsabfragen (Info-Teil und CRC32) bei der Firmware-Installation komplett abgeschaltet.

So kann ein Knotenbetreiber beliebige Images auf seinen Router spielen, vor allem aber das OpenWrt-Image. Man sollte hinterfragen, ob nicht wenigstens der Einsatz der CRC32-Prüfung sinnvoll ist, so dass zumindest eine kleine Hürde gegen das Einspielen einer ungewollten Datei aufgestellt wird.

Eine Konfigurationsdatei mit der entsprechenden Anpassung (CRC32 notwendig) ist verfügbar.[11]

3.3.5 Update aus einem bestehendem System

Ein OpenWrt-System bietet die Option an, ein neues Basissystem zu installieren, während das aktuelle noch ausgeführt wird. Achtung: Die angepassten **Einstellungen** unter `/etc` sind zuvor von Hand zu **sichern**!

Der Knotenbetreiber loggt sich dafür zunächst auf dem gewünschten Knoten ein und lädt das Firmware-Image des Basissystems im TRX-Format[WRT07b] über das Intra- oder Internet für die entsprechende Hardwareplattform herunter.

Anschließend kann das Image mit dem Kommando `mtd` geschrieben werden. Auf einzelnen

[9] Das RootFS des Netgear-Systems liegt während des Betriebes komplett im RAM und wird beim Booten mit den Dateien aus den lrp-Archiven (der „linux"-Partition) befüllt.

[10] http://downloads.openwrt.org/utils/wgt634u-upgrade.cfg

[11] http://sarwiki.informatik.hu-berlin.de/~mj/openwrt/wgt634u-crc32-mod.cfg

Plattformen (z.B. Meraki) muss der Linux-Kernel getrennt und *vor* dem RootFS eingespielt werden. Der Mitschnitt eines Updates ist in Listing 3.1 dargestellt.

Listing 3.1: Update des Basisimages auf einem WGT634U

```
1 $ tar czf - /etc | ssh user@client "cat > openwrt-config-backup.tar.gz"
2 $ cd /tmp
3 $ wget http://downloads.openwrt.org/kamikaze/7.07/brcm47xx-2.6/openwrt-brcm47xx
      -2.6-squashfs.trx
4 Connecting to downloads.openwrt.org [195.56.146.238:80]
5 openwrt-brcm47xx-2.6 100% |***************************|  1860 KB 00:00:00 ETA
6 $ mtd -e linux -r write openwrt-brcm47xx-2.6-squashfs.trx linux
7 Unlocking linux ...
8 Erasing linux ...
9 Writing from openwrt-brcm47xx-2.6-squashfs.trx to linux ...  [w]
10 Rebooting ...Restarting system.
```

3.4 Pakete für OpenWrt-Systeme

Um ein OpenWrt-System mit eigener Software zu erweitern, bietet es sich an, ein ipk-Paket zu erstellen und dieses zu verteilen. Details sind in [WRT07a] (Kapitel 2) zu finden.

Für spätere Updates kann man zusätzlich ein Paket-Repository einrichten. Hierzu speichert man die gewünschten Pakete auf einen Webserver und erstellt eine Auflistung der Paket-Metadaten unter dem Namen `Packages`.

4 Ausführung

Die bisher erstellten (Binär-)Programme wurden zwar für die konkrete Architektur (mips, x86) übersetzt, benötigen aber noch eine Ausführungsumgebung, um als Prozess gestartet zu werden. Auf den BRN-Knoten läuft zu diesem Zweck das universelle Betriebssystem: *Linux*.

4.1 Linux-Betriebssystem

Für ein lauffähiges Linux-System wird neben dem Systemkern (Kernel) ein Root-Dateisystem (RootFS) benötigt. Letzteres beinhaltet diverse System- und Anwendungsprogramme, deren Auswahl der jeweiligen Distribution obliegt.

4.1.1 Distributionen

Da der Betreiber eines Linux-Systems in der Regel die notwendigen Programme nicht selbst kompilieren möchte, existieren viele Linux-Distributionen, die die gewünschten Programme in Binärform und hilfreiche Skripte ausliefern. Bei den Systemprogrammen gelten die GNU-Programme [FSFa] als etabliert, so dass zum Teil auch von einem GNU/Linux-System gesprochen wird.

Im Bereich der eingebetteten Systeme – speziell bei WLAN-Routern – hat eine überschaubare Zahl an Linux-Distributionen Verbreitung erlangt. Erwähnenswert ist zum einen die auch im PC-Bereich bekannte Distribution Debian, die eine große Zahl an Hardwareplattformen unterstützt [Deb07a], zum anderen die etwas jüngere OpenWrt-Distribution, die eine gute Unterstützung für WLAN-Router bietet [WRTc] und im BRN eingesetzt wird.

4.1.2 Kernel

Der Linux-Kernel hat die Aufgabe, die vorhandene Hardware zu initialisieren und die dadurch bereitgestellten Ressourcen durch definierte Schnittstellen für die Prozesse (Programme in Ausführung) zu abstrahieren. Außerdem soll der Kernel die Prozesse voneinander isolieren[1] und den Zugriffe auf die genannten Ressourcen (gerecht) regeln.

Die Phase der Hardware-Initialisierung ist von Plattform zu Plattform verschieden und muss daher jeweils separat von den Kernel-Entwicklern gepflegt werden. Für viele Plattformen bringt der Vanilla-Kernel[2] bereits die notwendige Unterstützung mit, für andere (z.B. den WGT634U)

[1] Aus der Sicht des Prozesses, soll die Hardware exklusiv zur Verfügung stehen.
[2] Mit Vanilla-Kernel wird der offizielle Linux-Kernel, ohne spezielle Erweiterungen (durch Patches), bezeichnet.

werden Erweiterungen (Patches) von Drittprojekten (z.B. OpenWrt oder LinuxMIPS [MIPS]) benötigt.

So sind bei manchen Distributionen zwar Software-Pakete für die Architektur eines BRN-Knoten, aber kein lauffähiger Kernel verfügbar. Möchte man beispielsweise auf einem WGT634U die Debian-Distribution betreiben, kann man das RootFS sehr leicht mit den nötigen Binärprogrammen füllen[3], ist aber auf einen (selbst erstellten) Kernel von OpenWrt o.ä. angewiesen.

4.2 USB-Boot

Durch den am Netgear WGT634U vorhandenen USB-Anschluss werden zahlreiche Anwendungen ermöglicht. Eine interessant Anwendung, das Booten von einem USB-Speicher, soll im Folgenden vorgestellt werden.

4.2.1 Motivation

In der Praxis bietet diese Funktion einige Vorteile gegenüber dem Standard-Boot-Verhalten. So ist es möglich:

- Alternative Konfigurationen parallel auf demselben Knoten einzusetzen (z.B. Gateway statt BRN-Knoten – siehe Kapitel 5.2.3),
- Rettungsmaßnahmen auch ohne serielle Konsole (siehe Kapitel 5.2.1) durchzuführen,
- Ein größeres Root-Dateisystem (z.B. Debian mit glibc) zu nutzen.

4.2.2 Einschränkungen

Auf den bisher eingesetzten BRN-Knoten mit MIPS-CPU ist die Verwendung dahin gehend eingeschränkt, dass nur das RootFS, nicht aber der Kernel, von USB-Geräten geladen werden kann. Dies ist in der nicht vorhandenen USB-Unterstützung des Bootloaders (CFE) begründet. Zukünftig kann durch eine Portierung der kexec-Funktionalität [Nel04] auf die MIPS-Architektur eventuell Abhilfe geschaffen werden.

4.2.3 Variante 1: Anpassungen am Kernel

Exkurs: Initial Ramdisk

Mit Linux-Kernel 2.0 wurde eine Option eingeführt, die es ermöglicht, vor dem Mounten des eigentlichen Root-Dateisystems (nachfolgend *RootFS* genannt) eine so genannte „Initial Ramdisk“ (kurz: *initrd*) zu nutzen. Dabei handelt es sich um ein Dateisystem-Abbild (meist im Minix-Format),

[3] Hierzu bietet sich das Programm `debootstrap` [Deb07b] an.

welches in eine Ramdisk[4] kopiert und als vorläufiges RootFS eingebunden wird. Anschließend wird ein Programm mit dem Namen *linuxrc* gesucht und ausgeführt. Nach Beendigung von linuxrc wird der reservierte Speicher der Ramdisk wieder freigegeben und das eigentliche RootFS geladen.

Ein Linux-System auf diese Weise zu booten bietet einige Vorteile:

- Es können nachträglich **Treiber** (Kernel-Module) für Geräte **geladen werden**, deren Vorhandensein beim Erstellen des Kernels (beim Distributor) nicht vorab geprüft werden kann, ohne die aber das RootFS nicht nutzbar ist. (Zum Beispiel USB-Geräte oder Netzwerkkarten.) Somit können auch Treiber geladen werden, die nur abhängig von anderen Treibern funktionieren.

- Zusätzlich können auch **Closed-Source-Treiber** für RootFS-Geräte genutzt werden, die nur in Form von binären Kernel-Modulen vorliegen.

- Eine flexible Form der **Autokonfiguration** wird mit Initrds möglich. Auch andere Netzwerkdateisysteme als NFS (siehe Kapitel 5.3.2) und Konfigurationen abseits von DHCP (z.B. eine WLAN-Assoziation) können für das RootFS eingesetzt werden.

USB-Boot mit Initrd

Unter Zuhilfenahme einer solchen Initrd kann nun nach geeigneten USB-Geräten gesucht und das Mounten vorbereitet werden. Dazu muss der Wert der korrespondierenden Major- und Minornummern in die Pseudodatei `/proc/sys/kernel/real_root_dev` geschrieben werden.

Im Anhang A.1 auf Seite 38 ist ein entsprechendes Beispielprogramm dargestellt. Es lässt sich als statisch gelinktes linuxrc-Programm[5] nutzen.

USB-Boot mit Initramfs

Mit Linux 2.6 wurde das Konzept des Initialramfs [Sul04] [Lan05] eingeführt, welches die gleiche Funktionalität unterstützen soll, aber einige Vorteile bietet (Unabhängigkeit vom Bootloader, kein Entmounten und Mounten des RootFS beim Starten des Initprozesses mehr).

4.2.4 Variante 2: Anpassungen am RootFS

Neben der Anpassung des Linux-Kernels besteht durch kleine Änderungen an den Initskripten ebenfalls die Möglichkeit, das Dateisystem eines USB-Gerätes als RootFS zu nutzen.

[4] Eine Ramdisk ist ein virtuelles blockorientiertes Gerät, das (Sekundär-)Speicherblöcke auf Seiten (*Pages*) des Arbeitsspeichers abbildet.

[5] Somit werden keine zusätzlichen Bibliotheken benötigt.

Deshalb wird ein *zusätzliches* Skript nahezu am Anfang des Bootprozesses ausgeführt, das die notwendigen Treiber lädt (`usb-core`, `usb-storage`, `scsi-core`, `ext3` und weitere), nach passenden Geräten sucht, eines davon einbindet und das (bisher genutzte) RootFS verschiebt – genaueres kann man den Kommentaren im Skript (siehe Anhang A.2, Seite 41) entnehmen.

Der OpenWrt-Bootprozess erleichtert das Einbinden dieses Skriptes (/etc/preinit.usbboot), indem vor der Ausführung des üblichen Initprogramms (/sbin/init) ein Skript namens /etc/preinit aufgerufen wird. Hier kann das USB-Bootskript eingebunden werden[6]. Der Ablauf ist in Abbildung 4.1 schematisch dargestellt.

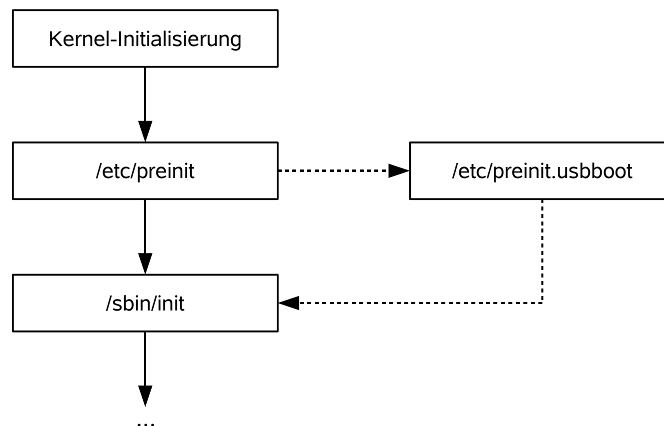


Abbildung 4.1: Programmaufruf beim Systemstart von OpenWrt

4.2.5 Vor- und Nachteile

Ein Anpassung des Kernels hat den Vorteil, dass die USB-Boot-Funktionalität auch dann gegeben ist, wenn das RootFS auf dem Flash-Speicher oder ein Teil der Initskripte defekt sein sollte, der Kernel aber noch gestartet wird. In diesem Fall könnte ein neues RootFS mit Hilfe des „USB-Linux-Systems“ eingespielt werden.

Die RootFS-Anpassung ermöglicht hingegen eine sehr einfache Aktivierung, Deaktivierung und Anpassung des USB-Boot-Skriptes – auch ohne Austausch des Firmware-Images. Außerdem beschränkt sich die Installation auf das Kopieren eines Skriptes und dem Editieren einer Datei. Die Kernel-Variante aber verlangt ein erhebliches Maß an Wissen über den Build-Prozess des Linux-Kernels.

Bei beiden Varianten ist zu beachten, dass Kernel-Module im Dateisystem des USB-Gerätes in jedem Fall zum installierten Kernel passen müssen. Dass heißt, eine aktualisierte Firmware zieht unter Umständen eine Aktualisierung der Kernel-Module nach sich.

[6]Zum aktuellen Zeitpunkt funktioniert das nachträgliche Einbinden nur mit `jffs2` als RootFS.

5 Testen und Debugging

Dieses Kapitel widmet sich der zentralen Frage: *Wie testen die Entwickler im BRN ihre Software und wie kann dieser Prozess unterstützt werden?*

5.1 Aspekte

In diesem Kapitel sollen Antworten auf zwei Fragen gegeben werden:

1. Wie kann die Software für Tests günstig verwaltet werden? (Wo sollen Programme, Bibliotheken und Log-Files bzw. Testdaten gespeichert werden?)

2. Wie gestaltet sich das Debugging im BRN-Umfeld?

5.2 BerlinRoofNet-Testbed

Zum Verständnis des folgenden Abschnitts ist eine Beschreibung der Umgebung, in dem die BRN-Knoten betrieben werden nötig. Diese lässt sich in mindestens 4 Klassen einteilen, wobei jede „niedrige“ Klasse eine Teilmenge der nächstgrößeren darstellt (z.B. kann jeder „Serial-connected Node“ auch ein Knoten am „Wired Backbone“ sein usw.).

5.2.1 Serial-connected Node

Selten, jedoch in manchen Fällen unabdingbar, ist die Situation, in der die BRN-Knoten direkt über eine so genannte *Serielle Konsole* mit einem Entwickler-Arbeitsplatz verbunden sind. (Die Beschränkung liegt in der Anzahl an vorhandenen Seriellen-Konsolen-Kabeln und Entwicklerrechnern mit serieller Schnittstelle.)

Der Anschluss einer Seriellen Konsole ermöglicht folgende Einsatzszenarien:

- **Ein Knoten** ist aufgrund von Konfigurations- oder Software-Fehlern nicht mehr über die üblichen Mechanismen (LAN, Wireless LAN) ansprechbar und **muss wieder hergestellt werden**.

- **Zum Testen und Debuggen** von Software, die im *Kernelspace* ausgeführt wird, ist ein (von Netzwerkverbindungen unabhängiger) Zugriff auf die so genannte „Kernel-Console“ notwendig. In Fällen, in denen diese Netzwerkunabhängigkeit keine Rolle spielt, kann auch das

Gespann *klogd/syslogd* oder das *Network console logging* benutzt werden. (CONFIG_NETCONSOLE, siehe auch [Mol02]).

- **Zum Anpassen von Linux-Distributionen** auf dem Knoten ist ein Zugriff teilweise bereits vor der Initialisierung der Netzwerkschnittstellen erforderlich.

Prinzipiell verfügen alle im BRN benutzten Plattformen über die Möglichkeit, auf diese Weise an einen Entwickler-Arbeitsplatz angeschlossen zu werden. Bei den Exemplaren des Linksys WRT54G(S) ist es aber notwendig, dass ein Anschluss auf die Platine gelötet wird – beim Netgear WGT634U und den x86-Boards ist dies nicht notwendig.

Wenn das serielle Kabel am Knoten angebracht wurde, muss noch das Terminal-Programm (z.B. screen, minicom oder Hyperterm) auf dem Entwickler-Arbeitsplatz eingerichtet werden. Zur Verbindung mit einem WGT634U (115200 b/s, 8N1) lässt sich dies wie folgt bewerkstelligen:

```
1 $ screen /dev/ttyS0 115200
```

5.2.2 Wired Backbone

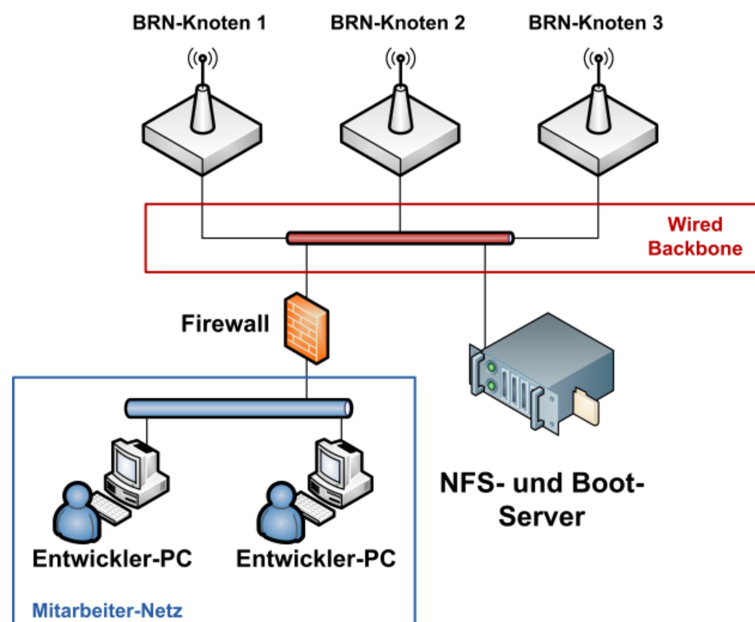


Abbildung 5.1: Schematischer Aufbau des Wired Backbone

Die wohl wichtigste Form der Kommunikation zwischen den Knoten und den Entwicklerrechnern erfolgt beim BRN über das lokale Netzwerk. Über dieses können alle angeschlossenen Knoten per Telnet bzw. SSH „aus der Ferne“ gesteuert werden. Dass heißt, es ist wie bei jedem UNIX-Rechner möglich, Programme zu starten, die Konfiguration anzupassen, neue Software einzuspielen und sogar eine komplett neue Firmware auf den Flash zu schreiben (siehe Kapitel 3.3.5).

Darüber hinaus können sehr einfach Netzwerkdateisysteme (z.B. NFS[1]) eingebunden und wie eine lokale Festplatte genutzt werden.

Zur besseren Handhabung werden die meisten dieser Knoten, wie in Abschnitt 5.3 genauer beschrieben, über das Netzwerk mit einem ausführbaren Kernel und dem RootFS (nfsroot) gestartet. Eine zentrale Datenspeicherung wird ebenfalls erleichtert.

Augrund dieser wichtigen Funktion soll das Netzwerk als **Wired Backbone** (dt. kabelgebundenes Rückgrat) bezeichnet werden. So sind aufgrund der genannten Vorteile fast alle vom Lehrstuhl betriebenen BRN-Knoten im Institutsgebäude an dieses Netzwerk angeschlossen. Abbildung 5.1 soll den Aufbau des Wired Backbones verdeutlichen.

Als „Testnetzwerk" wird es durch einen Firewall von der restlichen Lehrstuhl-Infrastruktur getrennt.

5.2.3 Wireless Backbone

Da es für manche Tests nicht möglich bzw. nicht praktikabel ist, die BRN-Knoten per Kabel mit dem Wired Backbone zu verbinden, besteht insbesondere bei Mobilitätstests (z.B. zur Bestimmung von Link-Qualitäten) der Wunsch, die gleiche Funktionalität mit Hilfe von WLAN-Technik nachzubilden bzw. auf diese Art einen direkten Zugang zum kabelgebundenen Netzwerk zu erhalten. Dieses Konzept trägt den Namen: **Wireless Backbone**.

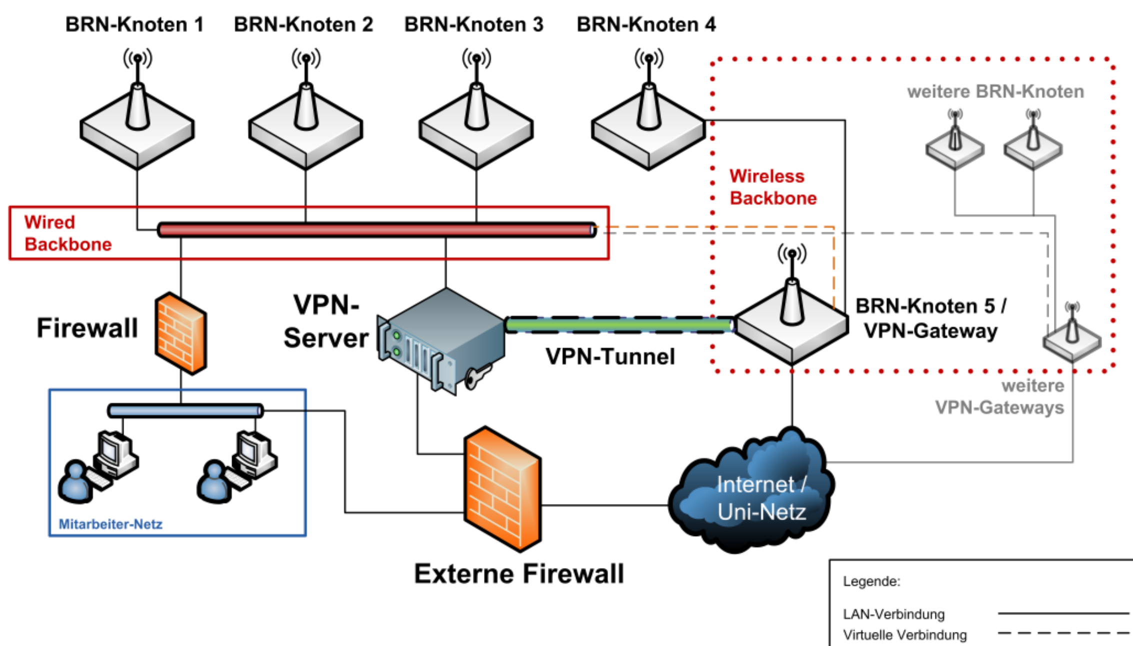


Abbildung 5.2: Schematischer Aufbau des Wireless Backbone

[1]Das *Network File System* (NFS) ist ein, unter UNIX, weit verbreitetes Netzwerkdateisystem, das die UNIX-Datei-Attribute (User- und Group-ID, 7-Bit-Rechtemaske, etc.) adäquat abbilden kann.

Ferner sollte der Einsatz transparent möglich sein, so dass Knoten, die zuvor am Wired Backbone betrieben wurden, ohne großen Aufwand auch ohne Kabelverbindung genutzt werden können.

Charakteristisch für diese Klasse von BRN-Knoten ist der fehlende physische Zugang zum geswitchten „Wired Backbone“. Dies betrifft praktisch alle „kabellosen“ Knoten auf dem Campus, aber auch jene Knoten, die bei Mitarbeitern und Studenten zu Hause betriebenen werden und über einen schnellen Internetzugang verfügen. Die Struktur ist in Abbildung 5.2 dargestellt.

Um die geforderte Eigenschaft der Transparenz umzusetzen, kann ein Layer-2-Tunnel[2] eingesetzt werden.

VPN-Bridge

Da man in der Regel keine unverschlüsselten Tunnel-Pakete über das Internet oder ein Funknetz übertragen möchte, kann für diesen Zweck auf VPN-Technologie[3] zurückgegriffen werden.

Aus Effizienzgründen wäre ein VPN-Tunnel auf Grundlage von IPsec [Ken98] sinnvoll. IPsec bietet aber nur eine IP-Tunnel-Funktionalität an (OSI-Layer 3), so dass ein weiteres Ethernet-over-IP-Tunnelprotokoll, wie EtherIP [Hou02] oder GRE [FLH00] auf dem IPsec-Tunnel aufsetzen muss. Leider bietet der Linux-Kernel bislang keine Unterstützung für EtherIP bzw. GRE mit Ethernet an.

Eine Alternative ist mit OpenVPN [Yon] verfügbar. Diese VPN-Implementierung arbeitet im Userspace und kommuniziert mit den zu schützenden Rechnern über eine virtuelle TUN- oder TAP-Netzwerkschnittstelle des Kernels.

Um die über den Tunnel transportierten Ethernet-Frames zwischen den eigentlichen LANs auszutauschen, wird die Ethernet-Bridging-Technologie des Linux-Kernels [Rad05] angewandt. Eine Beispiel-Konfiguration ist in Anhang B zu finden.

5.2.4 Outdoor Testbed

Jene BRN-Knoten ohne Verbindung zu einem *Sekundärnetz* sollen unter dem Namen **Outdoor Testbed** zusammengefasst werden. Bislang wurden auf diesem Gebiet wenig Erfahrungen gesammelt, deshalb muss diskutiert werden, in wie weit noch Forschungsarbeit notwendig ist.

5.2.5 Vor- und Nachteile

Wie in Kapitel 3.2.1 erläutert, ermöglichen die genannten Backbones eine schnelle Verteilung von zu testender Software und die zentrale Speicherung von Testergebnissen. Außerdem kann das BRN auf diese Weise schnell um neue Knoten erweitert werden.

Beim Wireless Backbone kann es jedoch zu einer konkurrierenden Nutzung des Mediums kommen, wenn der gleiche 802.11-Funkkanal genutzt wird, und somit können Testergebnisse verfälscht werden. Für den Betrieb des Wireless Backbone ist außerdem Wissen über VPN von Nöten.

[2] Dieser Tunnel transportiert Ethernet-Frames über ein anderes, meist IP-basiertes Netzwerk.

[3] Ein Virtual Private Network (VPN) verbindet private Netzwerke über ein öffentliches Netzwerk, z.B. das Internet.

Bei einem Outdoor-Szenario ist beispielsweise zu klären, wie die Tests angestoßen werden (vielleicht per SDP, siehe Kapitel 3.2.4) und wie die Ergebnisse gespeichert und gesammelt werden können.

Problem: Netzwerklatenz und TFTP

Bei der Nutzung von TFTP zum Laden des Linux-Kernels (siehe Kapitel 5.3.2) ist zu beachten, dass dieses Protokoll keine „Sammelquittungen" für die Bestätigung empfangener Datenpakete nutzt, sondern der Empfänger jedes Paket einzeln quittieren muss. Zudem ist die Blockgröße der Datenpakete auf 512 Bytes festgelegt. Bei einer größeren Netzwerklatenz, charakterisiert durch die *Round-Trip-Time*[4] (RTT), wird dadurch die maximal mögliche Datenrate proportional beeinflusst. Diese kann wie folgt berechnet werden:

$$\text{Datenrate} = \frac{\text{Blockgröße}}{\text{RTT}}.$$

Das Resultat kann daher stark von der prinzipiell möglichen Datenrate der Netzwerkverbindung abweichen. Beispielweise beträgt die TFTP-Datenrate bei einem typischen DSL-Anschluss mit 70 ms RTT:

$$\frac{512 \text{ Byte}}{0,07s} = 7314 \text{ Bytes/s}.$$

Der Bootprozess mit einem 2 MByte großen Kernel verzögert sich somit im Vergleich zum Wired Backone (1 ms RTT) um etwa 4-5 Minuten!

5.3 Common System Base

Für das Testen von verteilten Software-Systemen ist es, um Kompatibilitätsprobleme auszuschließen, meist erforderlich, dass die gleiche Software-Basis benutzt wird[5]. Um dieses Ziel zu erreichen, wird bei den BRN-Entwicklern ein etabliertes Verfahren benutzt, welches auch bei der Administration großer Rechnerpools eingesetzt wird – die Technik der *Linux Diskless Clients*.

5.3.1 Linux Diskless-Clients

(Linux-) Diskless-Clients [LDC] holen Programme und Dateien, kurzum das komplette Betriebssystem, von (mindestens) einem Dateiserver im Netzwerk. Im Gegensatz zu den funktional verwandten Thin Clients[6] [TC] laufen alle Anwendungsprogramme auf der lokalen Maschine, was zwar auf der einen Seite einen rechenstarken Terminalserver unnötig macht, aber auf der anderen

[4] Die RTT gibt die Verzögerungszeit an, die benötigt wird ein Netzwerkpaket von der Quelle zum Ziel und zurück zu übertragen.

[5] Dies sollte natürlich *nicht* für standardisierte Systeme gelten, bei denen die gleichen Schnittstellen verwendet werden, z.B. Netzwerk-Protokolle.

[6] Thin Clients haben meist nur wenig Hardware-Ressourcen und kümmern sich nur um die Ein- und Ausgabe. Die Software wird zu großen Teilen auf einem Terminal-Server ausgeführt.

Seite die Möglichkeiten eines Terminalservers (z.B. Umzug der Sitzung auf einen anderen Client) ungenutzt lässt.

Da solche Clients (normalerweise) keine Sekundärspeicher (z.B. Festplatten) und somit nach dem Ausschalten keinen individuellen Zustand besitzen, können sie im Wartungsfall leicht ausgetauscht werden und der Nutzer kann im Anschluss wie gewohnt weiterarbeiten. Außerdem verringern sich durch die fehlende Festplatte der Lärmpegel und die Stromaufnahme.

Da dieses Verfahren bereits ausführlich dokumentiert ist, soll es hier nur kurz umrissen und die Besonderheiten beim Einsatz der BRN-Knoten beschrieben werden. Weitergehende Informationen findet man zum Beispiel in [Suc06], [LTSP].

5.3.2 NFS-Root

Ein Linux-System – wie es auf den BRN-Knoten zum Einsatz kommt – benötigt für die korrekte Initialisierung im Wesentlichen zwei Komponenten:

1. ein ausführbares Kernel-Image,
2. ein Root-Dateisystem (welches vom Kernel unterstützt werden muss).

Nach der Erst-Initialisierung der Hardware (beispielsweise durch das BIOS in einem PC) sucht der Programmcode eines Boot-ROMs (der auch als Subroutine des Bootloaders implementiert sein kann) zunächst im lokalen Netzwerk nach einer freien IP-Adresse (und Informationen wie Netzmaske und Gateway). Früher kam dafür das *Reverse Address Resolution Protocol* (RARP) und *Boot Protocol* (BootP) zum Einsatz, heute verwendet man fast ausschließlich dessen Nachfolger, das *Dynamic Host Configuration Protocol* (DHCP). Ein Standard für Boot-ROMs existiert im PC-Bereich mit PXE [Int99], dessen Unterstützung sogar in Virtualisierungsprodukte wie zum Beispiel VMware Einzug gehalten hat.

Nach der IP-Konfiguration baut der Diskless Client eine Verbindung per *Trivial File Transfer Protocol* (TFTP) zum konfigurierten Bootserver[7] auf, um ein ausführbares Kernel-Image herunterzuladen und aus dem Arbeitsspeicher heraus zu starten. Dies ist beispielhaft für einen WGT634U in Listing 5.1 abgebildet.

Nachdem der Kernel die ihm bekannte Hardware eingerichtet hat, versucht er seinerseits die Netzwerkkarte zu konfigurieren. Dazu stehen im Kernelspace einige Optionen zur Verfügung (DHCP, BOOTP und RARP). Da im Netzwerk für den Transport des Kernels schon ein DHCP-Server existiert, bietet sich DHCP auch für die Kernel-Konfiguration an. Durch Anhängen von „`ip=dhcp`“ an die Kernel-Commandline (`CONFIG_CMDLINE` oder per Bootloader) wird der DHCP-Client im Kernel aktiviert.

Alternativ[8] kann man auch einen „echten“ Userspace-DHCP-Client oder andere Mechanismen in einer Initial-Ramdisk bzw. einem Initramfs starten (Vgl. Abschitt 4.2.3) und die Netzwerkhard-

[7] Der zuständige Bootserver kann ebenfalls über DHCP ermittelt oder muss manuell angegeben werden.

[8] Das ist in jedem Fall dann nötig, wenn sich der Netzwerkkarten-Treiber nicht monolithisch in den Linux-Kernel integrieren lässt.

ware konfigurieren lassen.

Listing 5.1: Bootvorgang eines WGT634U (CFE: DHCP & TFTP)

```
1 CFE> ifconfig eth0 -auto;boot -elf -tftp 192.168.4.3:vmlinux
2 Device eth0:  hwaddr 00-0F-B5-11-20-EE, ipaddr 192.168.3.62, mask 255.255.255.0
3        gateway 192.168.3.1, nameserver 192.168.2.20, domain sar.informatik.hu-
             berlin.de
4 Loader:elf Filesys:tftp Dev:eth0 File:192.168.4.3:vmlinux Options:(null)
5 ***************************
6 ****  MAC Client V1.0  ****
7 ***************************
8 et0macaddr value :flag =0 value=00-0f-b5-11-20-ee
9 et1macaddr value :flag =0 value=00-0f-b5-11-20-ef
10 MAC exist at least one
11 system ethernet mac exist and not default....
12 Skip mac client process.....
13 Loading: 0x80001000/2298176 0x80234000/360581 0x8028c085/167835 Entry at 0
     x80267000
14 Closing network.
15 et0: link down
16 Starting program at 0x80267000
17 Linux version 2.6.16.13-mips-noauthor-2 (mj@brn-suse093-5) (gcc version 3.4.6 (
     OpenWrt-2.0)) #1 Tue Feb 13 13:55:44 CET 2007
18 [...]
```

Listing 5.2: Bootvorgang eines WGT634U (Kernel: DHCP & NFS root)

```
1 [...]
2 b44: eth0: Link is up at 100 Mbps, full duplex.
3 b44: eth0: Flow control is off for TX and off for RX.
4 Sending DHCP requests ., OK
5 IP-Config: Got DHCP answer from 192.168.4.3, my address is 192.168.3.62
6 IP-Config: Complete:
7       device=eth0, addr=192.168.3.62, mask=255.255.255.0, gw=192.168.3.1,
8      host=wgt-20-ee, domain=sar.informatik.hu-berlin.de, nis-domain=sar,
9      bootserver=192.168.4.3, rootserver=192.168.4.3, rootpath=/openwrtroot
10 Looking up port of RPC 100003/2 on 192.168.4.3
11 Looking up port of RPC 100005/1 on 192.168.4.3
12 VFS: Mounted root (nfs filesystem).
13 [...]
```

Anschließend wird ein Root-Dateisystem (RootFS) ermittelt und versucht dieses einzubinden. Wurde in der Kernel-Konfiguration die Option „CONFIG_ROOT_NFS“ aktiviert und vom DHCP-Server[9] ein korrekter Pfad für das NFS-Share gesendet, mountet der Knoten den Pfad als Wurzel (/) ein und startet ein ausführbares Initprogramm, wie in Listing 5.2 dargestellt.

[9] Alternativ kann der Pfad auch statisch durch die Kernel-Commandline gesetzt werden.

Ab diesem Punkt verhält sich das System genau so, wie das eines Desktop-Rechners: Es werden die Initskripte abgearbeitet und der Nutzer erhält eine Login-Shell (an der seriellen Konsole) bzw. kann sich über das Netzwerk via Telnet oder SSH einloggen.

Eine Anpassung muss dennoch vorgenommen werden: In der Standard-Konfiguration versucht OpenWrt (wie andere Distributionen auch) jene Netzwerkschnittstelle zu konfigurieren, die als Grundlage der laufenden Prozesse dient. Das Initskript sägt sozusagen „den Ast ab, auf dem es sitzt“. Unter OpenWrt reicht es, in `/etc/config/network` die betroffenen Geräte auszukommentieren:

Listing 5.3: lst:wgtnet1

```
root@OpenWrt:/# cat /etc/config/network
#### VLAN configuration
#config switch eth0
#        option vlan0    "0 1 2 3 5*"
#        option vlan1    "4 5"


#### Loopback configuration
config interface loopback
        option ifname   "lo"
        option proto    static
        option ipaddr   127.0.0.1
        option netmask  255.0.0.0


#### LAN configuration
#config interface lan
#        option type     bridge
#        option ifname   "eth0.0"
#        option proto    static
#        option ipaddr   192.168.1.1
#        option netmask  255.255.255.0


#### WAN configuration
#config interface        wan
#        option ifname   "eth0.1"
#        option proto    dhcp
```

5.3.3 Homeverzeichnisse

Neben dem Root-Dateisystem bietet der NFS-Server außerdem Homeverzeichnisse für die BRN-Entwickler an. Sie ermöglichen es, dass jeder Entwickler seinen eigenen unabhängigen Datenbereich hat, der ebenfalls auf den BRN-Knoten im Wired und Wireless Backbone via nfs-mount zur Verfügung steht.

5.4 Debugging

Bei der Entwicklung umfangreicher Programme besteht häufig die Notwendigkeit, einen Debugger zum Auffinden von Programmierfehlern einzusetzen. Leider ist der Einsatz eines solchen auf den BRN-Knoten aufgrund der angesprochenen Ressourcenknappheit nur eingeschränkt, d.h. nur für „kleine“ Programme, möglich. So würde allein das Click-Binärprogramm mit den notwendigen Symbolen (GCC-Option: `-g`) schon 30 MB Hauptspeicher belegen – zum Vergleich: Die übliche Variante benötigt nur 3 MB.

5.4.1 Remote-Debugging

Um dennoch Programme wie den Click-Router zu debuggen, können die BRN-Entwickler auf das Konzept des **Remote-Debuggings** zurückgreifen [GDBb], [GDBc], [Gat]. Hierzu läuft auf dem Entwicklerrechner der eigentliche Debugger (z.B. der GDB[10] [GDBa]), während auf dem BRN-Knoten ein so genannter *Debugging-Agent* (auch Debugging-Stub genannt) läuft. Der Debugging-Agent im GDB-Umfeld heißt **gdbserver**.

5.4.2 Debugging-Beispiel

Zum Testen des Aufbaus der Debugging-Umgebung soll ein einfaches C-Programm dienen (siehe Ausgabe im Anhang C, Zeile 24-30). Dieses muss nun zunächst mit dem Cross-Compiler für die Zielplattform übersetzt (siehe Kapitel 2.2.2) und für den Knoten auf den NFS-Server kopiert werden:

```
1 $ kamikaze_7.07/staging_dir_mips*/bin/mips*-linux-gcc -static -g -o hello hello.c
2 $ scp hello nfs-server:/home/mj/
```

Im Anschluss muss auf dem BRN-Knoten der gdbserver mit dem soeben erstellten Programm `hello` gestartet werden:

```
1 root@OpenWrt:/# gdbserver :2000 /home/mj/hello
2 Process /home/mj/hello created; pid = 4544
3 Listening on port 2000
```

Nun kann wie in Anhang C versucht werden, den Fehler im Programm mit dem Debugger auf dem Entwickler-Rechner aufzuspüren. Eine Erleichterung (bei mehrfachem GDB-Aufruf) kann ein Skript bringen, in das der Entwickler alle GDB-Kommandos schreibt (hier: `gdb-script`).

[10] Es ist beim Cross-Debugging (Zielplattform $\neq$ Debugger-Plattform) darauf zu achten, den GDB aus der OpenWrt-Toolchain zu benutzen, z.B. `staging_dir_mipsel/bin/mipsel-linux-gcc`.

5.4.3 Tipps

Einige Hinweise können bei auftretenden GDB-Fehlermeldungen hilfreich sein:

- Der Debugger muss das Binärprogramm mit dem Kommando `file` einlesen, sonst heißt es: `No symbol table is loaded.`
- Zum „Starten“ des Programms muss `continue` statt des üblichen `run` verwendet werden, da das Programm bereits auf der Remote-Seite geladen wurde.
- Auf der Remote-Seite können auch „gestrippte“ Binärprogramme benutzt werden, solange auf der Debugger-Seite die ungestrippte Variante vorliegt.
- Beim Cross-Debugging von dynamisch gelinkten Programmen, ist die Option `solib-absolute-prefix` (Siehe Anhang C) für den Pfad der Bibliotheken anzugeben.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden Werkzeuge und Methoden zur Entwicklung von Softwarekomponenten für drahtlose Maschennetzwerke vorgestellt:

Die *OpenWrt-Toolchain* und das Build-System ermöglichen es, einen Cross-Compiler und andere wichtige Tools zur Software-Erstellung zu erzeugen und im Anschluss eigene Software als nativ ausfühbare Programme zu kompilieren.

Soll der generierte Binärcode (und andere Hilfsdateien) einer größeren Anzahl von BRN-Knoten zur Verfügung gestellt werden, kann ein Deployment mittels *NFS-Share*, *ipk-Paketen*, *SDP* und als *Firmware-Image* erfolgen.

Danach kann diese und andere Software unter Kontrolle eines *Linux-Systems* ausgeführt werden, welches auch eingeschränkt von USB-Medien booten kann, sofern die notwendigen Hardware-Voraussetzungen (z.B. ein USB-Anschluss) erfüllt sind.

Sollen umfangreiche Tests im drahtlosen Maschennetzwerk durchgeführt werden, können die Testbeds *Wired Backbone* und *Wireless Backbone* bei der Organisation der dafür benötigten Dateien helfen. Starten die BRN-Knoten sogar mittels *NFS-Root* von einem Bootserver, steht diesen sogar ein uniformes Linux-System zu Verfügung; Inkonsistenzen durch unterschiedliche Software-Versionen können so vermieden werden.

Treten Fehler in der Software auf, helfen sich Programmierer oft mit dem Einsatz eines Debuggers. *Remote-Debugging* (z.B. mit dem GDB und gdbserver) ermöglicht eine vergleichbare Technik auch für eingebettete Geräte.

6.2 Ausblick

In der Zukunft kann der Einsatz von Aufführungsumgebungen (Virtuelle Maschinen) wie Java oder Mono die Software-Entwicklung für drahtlose Maschennetzwerke erheblich vereinfachen. Dies gilt für die vorgestellten Entwicklungsprozesse: *Erstellung* (die Cross-Compilation fällt für die meisten Applikationen weg), *Deployment* (nur der Bytecode einer VM oder Skripte müssen noch verteilt werden, das Basissystem bleibt nach der Erst-Installation fix) und *Testen* (Software muss nicht mehr auf den Knoten selbst getestet werden; entsprechend ausgestattete Entwickler-PCs können diese Aufgabe übernehmen). Um den Einsatz solcher Umgebungen zu ermöglichen, müssen jedoch noch die Probleme des momentan enormen Ressourcenbedarfs geklärt werden.

A USB-Boot

A.1 Kernel-Variante

```
 /* ulinuxrc.c
 *
 * This program will be used as init program in the initial ramdisk (initrd)
 * of OpenWGT's linux kernel.
 *
 * Most WGT-users do not have a serial console, we will set the rootfs of
 * OpenWGT here. So it is not necessary to modify the kernel_args line at
 * the CFE (BIOS of WGT634U). (root=/dev/ram0 -> /dev/mtdblock3)
 *
 * Additionally it is possible to boot the root filesystem from an USB device,
 * if an working OpenWGT kernel is burned into the FLASH of the WGT.
 *
 * Also a so called ramdisk mode is possible.
 * Add "roottype=rd" to your kernel command line to enable it.
 *
 * (C) Mathias Jeschke 2006 <jeschke@informatik.hu-berlin.de>
 */

#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mount.h>
#include <sys/wait.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/unistd.h>

#define BUFSIZE 1024
#define INITIAL_REAL_ROOT_DEV "0x1f03"
#define INITIAL_REAL_ROOT_DEV_ALIAS "/dev/mtdblock/3"
#define RD_MODE_PARAMETER "roottype=rd"
#define RD_MODE_SCRIPT "/ramdisk-mode.sh"

char initial_real_root_dev[] = INITIAL_REAL_ROOT_DEV;  /* /dev/mtdblock3 (31,3) */
char *real_root_dev = initial_real_root_dev;
char *dev_list[] = { /* These devices will be probed */
            "/dev/scsi/host0/bus0/target0/lun0/disc", "0x0800",
            "/dev/scsi/host0/bus0/target0/lun0/part1", "0x0801",
            "/dev/scsi/host0/bus0/target0/lun0/part2", "0x0802",
            "/dev/scsi/host0/bus0/target0/lun0/part3", "0x0803",
            "/dev/scsi/host0/bus0/target0/lun0/part4", "0x0804",
            NULL };


void check_for_usb_dev() {

    int i;
    char* usb_mesg = " * Found USB device: ";
```

```
    for (i=0; dev_list[i]!=NULL; i+=2) { /* search for partition with ext3 fs */
        if ( (mount(dev_list[i], "/mnt", "ext3", MS_RDONLY, NULL) == 0) &&
                (access("/mnt/.version", R_OK) == 0) ) {
            real_root_dev = dev_list[i+1];
            write(1, usb_mesg, strlen(usb_mesg));
            write(1, dev_list[i], strlen(dev_list[i]));
            write(1, "\n", 1);
            umount("/mnt");
            break;
        }
    }
}


int check_for_rd_request() {

    char buf[BUFSIZE];
    int len;

    int fd = open("/proc/cmdline", O_RDONLY);
    len = read(fd, &buf, BUFSIZE);
    close(fd);

    return (strstr(buf, RD_MODE_PARAMETER) != NULL);
}


void prepare_rd_rootfs() {

    chdir("/flash");  chroot(".");

    /* Copying files from flash to ramdisk */
    execl(RD_MODE_SCRIPT, RD_MODE_SCRIPT, "prepare", NULL);
}


void set_root_dev() {

    /* Write new root device, since /dev/ram0 should not be replaced in kernel_args */
    int fd = open("/proc/sys/kernel/real-root-dev", O_WRONLY);
    write(fd, real_root_dev, strlen(real_root_dev));
    close(fd);
}


int main() {

    unsigned long mountflags = 0;
    int pid;

    sleep(2);    /* wait for USB initialization */

    write(1, "--- ulinuxrc started ---\n", 25);
    mount("proc", "/proc", "proc", mountflags, NULL);    /* We need procfs to set "real-root-dev" */

    check_for_usb_dev();

    /* Search for the "roottype=rd" substring in the kernel command line */
    if (real_root_dev == initial_real_root_dev) {
        /* Check for ramdisk mode */
        if (check_for_rd_request()) {
            write(1, " * Found ramdisk request!\n", 27);

            /* mount flash device to access the source files */
```

```
            write(1, " * Try to mount jffs2...         ", 34);
            if (mount(INITIAL_REAL_ROOT_DEV_ALIAS, "/flash", "jffs2", MS_RDONLY, NULL) == 0) {
                write(1, "succeeded.\n", 11);

                write(1, " * Check for RD mode script...   ", 34);
                if (access("/flash"RD_MODE_SCRIPT, X_OK) == 0) {
                    write(1, "succeeded.\n", 11);

                    /* mount tmpfs device to write the destination files */
                    write(1, " * Try to mount tmpfs...         ", 34);
                    if (mount("/dev/shm", "/flash/mnt", "tmpfs", 0, NULL) == 0) {
                        write(1, "succeeded.\n", 11);

                        if ((pid = vfork()) == 0) { /* Child */
                            prepare_rd_rootfs();
                            exit(0); /* Exit child, if exec() fails */
                        }
                        else if (pid > 0) {
                            wait(NULL);
                            write(1, " * mv /flash/mnt --> /newroot\n", 30);
                            mount("/flash/mnt", "/newroot", "tmpfs", MS_MOVE, NULL);
                            umount("/flash");

                            write(1, " * Pivoting rootfs...\n", 23);
                            chdir("/newroot");  pivot_root(".", "mnt");
                            mount("/mnt/proc", "/proc", "proc", MS_MOVE, NULL);
                            write(1, "--- ulinuxrc done ---\n", 22);
                            execl(RD_MODE_SCRIPT, RD_MODE_SCRIPT, "start", NULL);
                        }
                        else {
                            /* Fork failed */
                            write(1, " * Can't fork! Resetting...\n", 28);
                            umount("/flash/mnt");
                            umount("/flash");
                            /* Continue with normal operation below */
                        }
                    }
                    else {
                        write(1, "failed.\n", 8);
                        umount("/flash");
                        /* Continue with normal operation below */
                    }
                }
                else {    /* File not found or not executable */
                    write(1, "failed.\n", 8);
                    umount("/flash");
                    /* Continue with normal operation below */
                }
            }
            else {
                write(1, "failed.\n", 8);
                /* Continue with normal operation below */
            }
        }
    }

    set_root_dev();

    umount("/proc");
    write(1, "--- ulinuxrc done ---\n", 22);
    return 0;
}
```

A.2 RootFS-Variante

```
# /etc/preinit.usbboot
#
# If you want to enable USB rootfs boot,
# copy this file to /etc/preinit.usbboot (on your jffs2 rootfs)
# and insert the following line before "exec /sbin/init":
#     . /etc/preinit.usbboot
#
# (C) Mathias Jeschke 2007 <openwrt@majes.de>
#
# url: https://openwrt.majes.de/base-files/etc/preinit.usbboot
# $Id: preinit.usbboot 18 2007-09-04 23:34:18Z mj $

# Redirecting stdout to console
exec >/dev/console
echo "- preinit.usbboot -"

# Loading needed kernel modules
for m in usbcore ohci-hcd ehci-hcd scsi_mod sd_mod usb-storage jbd ext3; do
    insmod $m
done

# Waiting for USB initialization
sleep 8

echo "=> Trying to mount USB device (with ext3 fs)"

# Preparing device nodes
PREFIX="/dev/scsi/host0/bus0/target0/lun0"
if ! grep -q devfs /proc/mounts; then
    # Generating device nodes
    mkdir -p $PREFIX
    mknod $PREFIX/disc b 8 0
    for i in $(seq 1 4); do
        mknod $PREFIX/part$i b 8 $i
    done
fi

# Trying to mount a known fs type
mount $PREFIX/disc /mnt || \
mount $PREFIX/part1 /mnt || \
mount $PREFIX/part2 /mnt || \
mount $PREFIX/part3 /mnt || \
mount $PREFIX/part4 /mnt || \
    exec /sbin/init

# Checking for minimal requirements on usb device
```

```
[ -x /mnt/sbin/init ] && \
[ -d /mnt/dev ] && \
[ -d /mnt/proc ] && \
[ -d /mnt/tmp ] || exec /sbin/init

echo  "=> Found USB boot device at $(awk '/mnt/ print $1 ' /proc/mounts)"
if [ -x /mnt/etc/preinit.usb ]; then
    echo "=> Exec preinit.usb at /mnt"
    # Loading the following stuff from usb script
    exec sh -x /mnt/etc/preinit.usb >/dev/console 2>&1
fi

# Mounting device + virtual file systems for new (usb) root
if grep -q devfs /proc/filesystems; then
    mount -t devfs none /mnt/dev
else
    mount -o bind /dev /mnt/dev
fi
mount -t devpts none /mnt/dev/pts

# Unmounting old "automounted" usb devices
umount /tmp/sda* 2>&-

# Unmounting/Moving old filesystems
mount -o remount,ro /
umount /dev/pts
umount /dev
mount -o move /sys /mnt/sys
mount -o move /tmp  /mnt/tmp
mount -o move /proc /mnt/proc

# Switching to new rootfs
cd /mnt
pivot_root . mnt

# We need a new program file for our process
# (the old one still has open libs on /mnt/lib/...)
echo "- init (usb) -"
exec chroot . sh -c 'umount /mnt/dev; umount /mnt; exec /sbin/init' <dev/console >dev/console 2>&1
```

B VPN-Gateway

```
# /etc/openvpn/vpn-bridge-server.ovpn
#
# Beschreibende Kommentare mit #
# Auskommentierte Optionen mit ;

cd /etc/openvpn     # So koennen die Dateinamen relativ angegeben werden.

port 1194           # Port auf dem OpenVPN lauscht
;proto udp          # UDP oder TCP? (UDP ist Standard)
dev tap             # TUN- oder TAP-Device

# SSL-Authentifizierungsoptionen
# key  - privater Schluessel
# cert - eigenes Zertifikat (von CA signiert)
# ca   - CA-Zertifikat (zur Verifikation)
key vpn-bridge-server.key
cert vpn-bridge-server.cert
ca ca.cert

# Symmetrischer Schluessel fuer HMAC (gegen DoS)
# Muss auf allen Knoten gleich sein, erstellen mit
#   openvpn --genkey --secret preshared.key
tls-auth preshared.key 0

dh dh1024.pem       # Diffie-Hellman-Parameter koennen erstellt werden mit:
                    #   openssl dhparam -out dh1024.pem 1024
mode server
tls-server

# DHCP-Client fuegt neue default-route hinzu -> Zyklus
push "route remote_host 255.255.255.255 net_gateway"

keepalive 10 60     # Senden von Heartbeat-Paketen (Aufrechterhaltung der "Verbindung")

comp-lzo            # Komprimiere Pakete

user nobody         # Reduzierung der Prozessprivilegien
group nogroup
persist-key
persist-tun

verb 0              # Kaum Infomeldungen (Hochsetzen fuer Debug)

float               # Falls Client-IP-Adressen waehrend der Verbindung sich
                    # aendern sollten (Dial-Up-Reconnect)

up ./up-script.sh   # Skript zum automatischen Einrichten der Bridge
                    # Inhalt:
                    ##!/bin/sh
                    #
                    #brctl addif br-lan $1; ifconfig $1 up
```

C Remote-GDB-Skript

```
$ cat gdb-script
file hello
# Auskommentieren bei dynamisch gelinkten Programmen
#set solib-absolute-prefix /tmp/kamikaze_7.07/staging_dir_mipsel/
list
target remote 192.168.3.100:2000
break 4
break 5
cont
print out
cont
print out
cont

$ kamikaze_7.07/staging_dir_mipsel/bin/mipsel-linux-gdb < gdb-script
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i586-suse-linux --target=mipsel-linux-uclibc".
(gdb) Reading symbols from /tmp/hello...done.
(gdb) 1 int main(void) {
2
3               char *out = "Dies ist ein Test!\n";
4               out = 0;
5               printf("%c", *out);
6               return 0;
7       }
(gdb) Remote debugging using 192.168.3.100:2000
0x00400140 in _ftext ()
(gdb) Breakpoint 1 at 0x400300: file hello.c, line 4.
(gdb) Breakpoint 2 at 0x400304: file hello.c, line 5.
(gdb) Continuing.

Breakpoint 1, main () at hello.c:4
4               out = 0;
(gdb) $1 = 0x402430 "Dies ist ein Test!\n"
(gdb) Continuing.

Breakpoint 2, main () at hello.c:5
5               printf("%c", *out);
(gdb) $2 = 0x0
(gdb) Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0040030c in main () at hello.c:5
5               printf("%c", *out);
```

Literaturverzeichnis

[802.11] IEEE, Standards for Information Technology: *IEEE 802.11, 1999 Edition (ISO/IEC 8802-11: 1999) – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999. http://standards.ieee.org/getieee802/download/802.11-1999.pdf

[ASU86] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers*. Addison Wesley, 1986. – ISBN 0201100886

[BRN] SYSTEMS ARCHITECTURE GROUP: *Berlin Roof Net (BRN)*. http://berlinroofnet.de/

[CLKa] *Click: Example Configurations*. http://www.read.cs.ucla.edu/click/examples

[CLKb] *Click: Tutorial Problem Set 1*. http://www.read.cs.ucla.edu/click/tutorial1

[Deb07a] DEBIAN: *Portierungen*. http://www.debian.de/ports/. Version: 2007

[Deb07b] DEBIAN INSTALL SYSTEM TEAM: Debootstrap – Erststart eines Debian Basis-Systems. (2007). http://packages.debian.org/stable/admin/debootstrap

[Dro97] DROMS, R.: *RFC 2131: Dynamic Host Configuration Protocol*, 1997. http://tools.ietf.org/html/rfc2131

[FLH00] FARINACCI, D. ; LI, T. ; HANKS, S.: *RFC 2784: Generic Routing Encapsulation (GRE)*, 2000. http://tools.ietf.org/html/rfc2784

[FSFa] FREE SOFTWARE FOUNDATION: *Coreutils – GNU core utilities*. http://www.gnu.org/software/coreutils/

[FSFb] FREE SOFTWARE FOUNDATION: *GNU C Library*. http://www.gnu.org/software/libc/

[Gat] GATLIFF, Bill: An Overview of Remote Debugging. http://venus.billgatliff.com/node/2

[GDBa] *GDB: The GNU Project Debugger*. http://www.gnu.org/software/gdb/gdb.html

[GDBb] *Debugging with GDB: 16. Specifying a Debugging Target*. http://sourceware.org/gdb/current/onlinedocs/gdb_17.html

[GDBc] *Debugging with GDB: 17. Debugging Remote Programs*. http://sourceware.org/gdb/current/onlinedocs/gdb_18.html

[Hou02] HOUSLEY, R.: *RFC 3378: EtherIP: Tunneling Ethernet Frames in IP Datagrams*, 2002. http://tools.ietf.org/html/rfc3378

[Int99] INTEL CORPORATION: *Preboot Execution Environment (PXE) Specification*, 1999. http://www.pix.net/software/pxeboot/archive/pxespec.pdf

[Ipkg] *ipkg - the Itsy Package Management System*. http://handhelds.org/moin/moin.cgi/Ipkg

[JFFS2] *JFFS2: The Journalling Flash File System, version 2*. http://sourceware.org/jffs2/

[Ken98] KENT, S.: *RFC 2401: Security Architecture for the Internet Protocol*, 1998. http://tools.ietf.org/html/rfc2401

[KMC+00] KOHLER, Eddie ; MORRIS, Robert ; CHEN, Benjie ; JANNOTTI, John ; KAASHOEK, Frans M.: The click modular router. In: *ACM Trans. Comput. Syst.* 18 (2000), August, Nr. 3, 263–297. http://dx.doi.org/10.1145/354871.354874. – DOI 10.1145/354871.354874. – ISSN 0734–2071

[Kra02] KRASNYANSKY, Maxim: Universal TUN/TAP device driver. (2002). http://lxr.linux.no/source/Documentation/networking/tuntap.txt

[Lan05] LANDLEY, Rob: Introducing initramfs, a new model for initial RAM disks. (2005). http://linuxdevices.com/articles/AT4017834659.html

[LDC] *Linux Diskless Clients*. http://www.ks.uni-freiburg.de/projekte/ldc/

[LEAF] *Linux Embedded Appliance Firewall*. http://leaf.sourceforge.net/

[Lin] *The Linux Kernel Archives*. http://kernel.org/

[LTSP] LINUX TERMINAL SERVER PROJECT: *LTSP Documentation*. http://wiki.ltsp.org/twiki/bin/view/Ltsp/Documentation

[MAD] *Madwifi: Architecture*. http://madwifi.org/wiki/Architecture

[MIPS] *LinuxMIPS: Main Page*. http://www.linux-mips.org/wiki/Main_Page

[Moc87a] MOCKAPETRIS, P.: *RFC 1034: Domain names - concepts and facilities*, 1987. http://tools.ietf.org/html/rfc1034

[Moc87b] MOCKAPETRIS, P.: *RFC 1035: Domain names - implementation and specification*, 1987. http://tools.ietf.org/html/rfc1035

[Mol02] MOLLNAR, Ingo: Linux Kernel sources: netconsole.txt. (2002). http://lxr.linux.no/source/Documentation/networking/netconsole.txt

[MONOa] *Mono: Main Page*. http://www.mono-project.com/

[MONOb] *SarWiki: Mono for BRN Hardware Platform*. http://sarwiki.informatik.hu-berlin.de/S-07S-09

[MTRK] *Metrik – Modellbasierte Entwicklung von Technologien für selbstorganisierende dezentrale Informationssysteme im Katastrophenmanagement*. http://metrik.informatik.hu-berlin.de/

[Nel04] NELLITHEERTHA, Hariprasad: Reboot Linux faster using kexec. (2004). http://www.ibm.com/developerworks/linux/library/l-kexec.html

[Pcap] *TCPDUMP public repository*. http://www.tcpdump.org/

[Rad05] RADTKE, Nils: Ethernet Bridge + netfilter Howto. (2005). http://www.tldp.org/HOWTO/Ethernet-Bridge-netfilter-HOWTO.html

[Sqshfs] *SQUASHFS - A squashed read-only filesystem for Linux*. http://squashfs.sourceforge.net/

[Suc06] SUCHODOLETZ, Dirk v.: OpenSLX Linux Diskless Clients - Aufbau und Betrieb von Diskless X-Stations. (2006). http://www.ks.uni-freiburg.de/download/projects/ldc/openslx-doku.pdf

[Sul04] SULLIVAN, Bryan O.: Early userspace support. (2004). http://lxr.linux.no/source/Documentation/early-userspace/README

[TC] Thin Clients und Server Based Computing. http://www.thin-client.info/

[TW06] TANENBAUM, Andrew S. ; WOODHULL, Albert S.: *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, 2006. – ISBN 0131429388

[uClb] *uClibc*. http://uclibc.org/

[Ull06] ULLENBOOM, Christian: *Java ist auch eine Insel. Programmieren mit der Java Standard Edition Version 5 / 6*. Galileo Press, 2006. – ISBN 3898428389

[Wie06] WIEDEMANN, Bernhard: *Development of a Software Distribution Platform for the BerlinRoofNet*, Diplomarbeit, Januar 2006. http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-01/SAR-PR-2006-01_.pdf

[WRTa] *OpenWrt*. http://openwrt.org/

[WRTb] *OpenWrt: packages*. `https://svn.openwrt.org/openwrt/packages/`

[WRTc] *OpenWrt: TableofHardware*. `http://toh.openwrt.org/`

[WRT07a] OpenWrt: Kamikaze Documentation. (2007). `http://downloads.openwrt.org/kamikaze/docs/openwrt.pdf`

[WRT07b] *OpenWrt: TRX vs. BIN*. `http://wiki.openwrt.org/OpenWrtDocs/Installing`. Version: 2007

[Yon] YONAN, James: *OpenVPN - An Open Source SSL VPN Solution*. `http://openvpn.net/`

Reports published by Humboldt University Berlin, Computer Science Department, Systems Architecture Group.

1. SAR-PR-2005-01: Linux-Hardwaretreiber für die HHI CineCard-Familie. Robert Sperling. 37 Seiten.
2. SAR-PR-2005-02, NLE-PR-2005-59: State-of-the-Art in Self-Organizing Platforms and Corresponding Security Considerations. Jens-Peter Redlich, Wolf Müller. 10 pages.
3. SAR-PR-2005-03: Hacking the Netgear wgt634u. Jens-Peter Redlich, Anatolij Zubow, Wolf Müller, Mathias Jeschke, Jens Müller. 16 pages.
4. SAR-PR-2005-04: Sicherheit in selbstorganisierenden drahtlosen Netzen. Ein Überblick über typische Fragestellungen und Lösungsansätze. Torsten Dänicke. 48 Seiten.
5. SAR-PR-2005-05: Multi Channel Opportunistic Routing in Multi-Hop Wireless Networks using a Single Transceiver. Jens-Peter Redlich, Anatolij Zubow, Jens Müller. 13 pages.
6. SAR-PR-2005-06, NLE-PR-2005-81: Access Control for off-line Beamer – An Example for Secure PAN and FMC. Jens-Peter Redlich, Wolf Müller. 18 pages.
7. SAR-PR-2005-07: Software Distribution Platform for Ad-Hoc Wireless Mesh Networks. Jens-Peter Redlich, Bernhard Wiedemann. 10 pages.
8. SAR-PR-2005-08, NLE-PR-2005-106: Access Control for off-line Beamer Demo Description. Jens Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge. 18 pages.
9. SAR-PR-2006-01: Development of a Software Distribution Platform for the Berlin Roof Net (Diplomarbeit / Masters Thesis). Bernhard Wiedemann. 73 pages.
10. SAR-PR-2006-02: Multi-Channel Link-level Measurements in 802.11 Mesh Networks. Mathias Kurth, Anatolij Zubow, Jens Peter Redlich. 15 pages.
11. SAR-PR-2006-03, NLE-PR-2006-22: Architecture Proposal for Anonymous Reputation Management for File Sharing (ARM4FS). Jens Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge, Torsten Dänicke. 20 pages.
12. SAR-PR-2006-04: Self-Replication in J2me Midlets. Henryk Plötz, Martin Stigge, Wolf Müller, Jens-Peter Redlich. 13 pages.
13. SAR-PR-2006-05: Reversing CRC – Theory and Practice. Martin Stigge, Henryk Plötz, Wolf Müller, Jens-Peter Redlich. 24 pages.
14. SAR-PR-2006-06: Heat Waves, Urban Climate and Human Health. W. Endlicher, G. Jendritzky, J. Fischer, J.-P. Redlich. In: Kraas, F., Th. Krafft & Wang Wuyi (Eds.): Global Change, Urbanisation and Health. Beijing, Chinese Meteorological Press.
15. SAR-PR-2006-07: 无线传感器网络研究新进展 (State of the Art in Wireless Sensor Networks). 李刚 (Li Gang), 伊恩斯•彼得•瑞德里希 (Jens Peter Redlich)

16. SAR-PR-2006-08, NLE-PR-2006-58: Detailed Design: Anonymous Reputation Management for File Sharing (ARM4FS). Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge, Christian Carstensen, Torsten Dänicke. 16 pages.
17. SAR-PR-2006-09, NLE-SR-2006-66: Mobile Social Networking Services Market Trends and Technologies. Anett Schülke, Miquel Martin, Jens-Peter Redlich, Wolf Müller. 37 pages.
18. SAR-PR-2006-10: Self-Organization in Community Mesh Networks: The Berlin RoofNet. Robert Sombrutzki, Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 11 pages.
19. SAR-PR-2006-11: Multi-Channel Opportunistic Routing in Multi-Hop Wireless Networks. Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 20 pages.
20. SAR-PR-2006-12, NLE-PR-2006-95 Demonstration: Anonymous Reputation Management for File Sharing (ARM4FS). Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen, Torsten Dänicke. 23 pages.
21. SAR-PR-2006-13, NLE-PR-2006-140 Building Blocks for Mobile Social Networks Services. Jens-Peter Redlich, Wolf Müller. 25 pages.
22. SAR-PR-2006-14 Interrupt-Behandlungskonzepte für die HHI CineCard-Familie. Robert Sperling. 83 Seiten.
23. SAR-PR-2007-01 Multi-Channel Opportunistic Routing. Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 10 pages. IEEE European Wireless Conference, Paris, April 2007.
24. SAR-PR-2007-02 ARM4FS: Anonymous Reputation Management for File Sharing. Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen, 10 15 pages.
25. SAR-PR-2007-03 DistSim: Eine verteilte Umgebung zur Durchführung von parametrisierten Simulationen. Ulf Hermann. 26 Seiten.
26. SAR-SR-2007-04 Architecture for applying ARM in optimized pre-caching for Recommendation Services. Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen. 29 pages.
27. SAR-PR-2007-05 Auswahl von Internet-Gateways und VLANs im Berlin RoofNet. Jens Müller. 35 Seiten.