Studienarbeit

Dynamische Protokoll-Erweiterung für drahtlose Maschennetzwerke

Dirk Rempe

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

Gutachter:

.....

.....

Betreuer: Robert Sombrutzki

Berlin, 27. September 2018

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ziel . . . 1
1.2 Aufbau der Arbeit . . . 1

2 Grundlagen **2**
2.1 OSI Referenzmodell . . . 2
2.2 Frameworks . . . 3
2.3 Nachladen von Programmcode . . . 4

3 Implementierung **6**
3.1 Auswahl des Frameworks und der Programmiersprache . . . 6
3.2 Umsetzung . . . 7
3.3 Einbindung in Click . . . 9

4 Evaluation **10**
4.1 Zunehmende Parameterzahl mit leerem Funktionskörper . . . 12
4.2 Zunehmende Parameterzahl mit minimalem Funktionskörper . . . 13
4.3 Zunehmende Anzahl an Funktionen . . . 13
4.4 Zunehmende Länge des Funktionskörpers . . . 14
4.5 Ergebnis . . . 15

5 Zusammenfassung und Ausblick **16**

Abbildungsverzeichnis **17**

Literaturverzeichnis **18**

Kapitel 1

Einleitung

Netzwerke und insbesondere drahtlose Netzwerke haben in den vergangenen Jahren immer mehr an Bedeutung gewonnen. In diesem Zuge sind zahlreiche Protokolle und Standards zur kontrollierten, effizienten und sicheren Datenübertragung entstanden. Die Kommunikation ist dabei komplex und vielschichtig. Um dieses adäquat abzubilden, ist z.B. das OSI-Schichtenmodell entstanden. Es ordnet jedem Protokoll/ Dienst eine von sieben Schichten(Layern) zu, beginnend bei der Bitübertragungs-, über die Transport-, bis zur den Anwendungsschicht [Abbildung 2.1]. Dabei nehmen die Anforderungen durch die wachsende Anzahl mobiler Endgeräte wie Handys, Tablets, etc. stetig zu. Um den wachsenden Anforderungen gerecht zu werden, müssen z.B. Router, Firewalls, Virenscanner fortlaufend weiterentwickelt werden. Die Umsetzung der Protokollstacks ist aber meistens recht starr und die Funktionalitäten sind sehr statisch implementiert, was das Weiterentwickeln bestehender oder die Entwicklung neuer Protokolle erschwert. An diesem Punkt möchte diese Arbeit anknüpfen.

1.1 Ziel

Das Ziel der Arbeit ist es einen Überblick zu gewinnen, wie Quellcode dynamisch in ein laufendes Programm nachgeladen werden kann. Des Weiteren ist die Umsetzung in möglichen Netzwerkanwendungen/-Frameworks abzuwiegen und schließlich eine Implementierung in einem Framework umzusetzen und auf seine Performance hin zu untersuchen.

1.2 Aufbau der Arbeit

Im Kapitel 2 Grundlagen werde ich auf das OSI-Netzwerkreferenzmodell eingehen. In diesem Zusammenhang ein paar netzwerkbasierte Anwendungen, Frameworks betrachten und in welcher Form und wie sich Quellcode dynamisch in ein laufendes Programm einbinden lässt.
Im 3. Kapitel gehe ich auf die Wahl des Frameworks und die Programmiersprache ein, die ich für meine Implementierung verwende. Anschließend gehe ich auf einige Implementierungsdetails bzgl. des dynamischen Nachladens von Code und die Anbindung an Click ein.
Im letzten Kapitel 4 (Evaluation) wird untersucht, in wieweit das Kompilieren und Einbinden des Quellcodes, anhand der hier umgesetzten Implementierung, skaliert und wodurch die Zeit beeinflusst wird.

Kapitel 2

Grundlagen

2.1 OSI Referenzmodell

Die Kommunikation in Netzwerken ist komplex. Die dabei zu lösenden Probleme reichen von Fragen der elektronischen Signalübertragung über die Regelung der Reihenfolge in der Kommunikation bis hin zu abstrakteren Aufgaben, die sich innerhalb der kommunizierenden Anwendungen ergeben. Um der Vielzahl an Aufgaben gerecht zu werden, hat man z.B. das OSI-Schichtenmodel [Abbildung 2.1] eingeführt, das die unterschiedlichen Kommunikationsabläufe in sieben Schichten aufteilt. In jeder einzelnen Schicht werden die Anforderungen separat umgesetzt. Jede Schicht stellt bestimmte Dienste zur Verfügung, die die darüber liegenden Schichten nutzen können. Dabei bauen die Dienste einer oberen Schicht auf denen der darunter liegenden Schichten auf. Die Kommunikation zwischen zwei Instanzen (Sender und Empfänger) sollte immer zwischen Diensten derselben Schicht ablaufen. Eine solche Kommunikation verlangt feste Regeln, welche in Form von Protokollen abgebildet werden. Ein Protokoll ermöglicht eine horizontale Verbindung zwischen zwei Instanzen auf derselben Schicht.
Die Abbildung [2.1] zeigt die sieben Schichten des OSI-Modells mit der Zuordnung einiger Beispielprotokolle.

OSI-Schicht		Einordnung	DoD-Schicht	Einordnung	Protokollbsp.	Einheiten	Kopplungselemente
7	Anwendungen (Application)	Anwendungs-orientiert			HTTP FTP HTTPS		
6	Darstellung (Presentation)		Anwendung		SMTP XMPP DNS	Daten	Gateway, Content-Switch,
5	Sitzung (Session)			Ende zu Ende (Multihop)	LDAP NCP		Proxy, Layer-4-7-Switch
4	Transport (Transport)	Transport-orientiert	Transport		TCP UDP SCTP SPX	TCP = Segmente UDP = Datagramme	
3	Vermittlung-/Paket (Network)		Internet		ICMP IGMP IP IPsec IPX	Pakete	Router, Layer-3-Switch
2	Sicherung (Data Link)		Netzzugriff	Punkt zu Punkt	Ethernet Token Ring FDDI	Rahmen (Frames)	Bridge, Layer-2-Switch
1	Bitübertragung (Physical)				MAC ARCNET	Bits, Symbole, Pakete	Netzwerkkabel, Repeater, Hub

Abbildung 2.1: Das OSI-Modell im Überblick [15]

Die sieben Schichten bilden folgende Aufgaben ab:

7. Schicht	**Anwendung**	-Funktionen für Anwendungen, sowie die Dateneingabe und -ausgabe
6. Schicht	**Darstellung**	-Umwandlung der systemabhängigen Daten in ein unabhängiges Format
5. Schicht	**Sitzung**	-Steuerung der Verbindungen und des Datenaustauschs
4. Schicht	**Transport**	-Zuordnung der Datenpakete zu einer Anwendung
3. Schicht	**Vermittlung**	-Routing der Datenpakete zum nächsten Knoten
2. Schicht	**Sicherung**	-Segmentierung der Pakete in Frames und Hinzufügen von Prüfsummen
1. Schicht	**Bitübertragung**	-Umwandlung der Bits in ein zum Medium passendes Signal und physikalische Übertragung

Es gibt also unterschiedlichste Protokolle, die den einzelnen Schichten zugeordnet sind. Nehmen wir als Beispiel das TCP-Protokoll (Transmission Control Protocol [18]) heraus, welches der Transportschicht (OSI-Schicht 4) zugeordnet ist. Es ist ein zuverlässiges, verbindungsorientiertes, paketvermitteltes Transportprotokoll und legt fest, auf welche Art und Weise Daten zwischen zwei Netzwerkkomponenten ausgetauscht werden. Einige Teile des Protokolls haben starre Vorgaben, wie der Verbindungsaufbau und -abbau. Andere Teile bieten hingegen noch Entwicklungsmöglichkeiten, z.B. die Überlaststeuerung, also was passiert, wenn Pakete aufgrund zu hoher Netzwerklast verloren gehen. Dafür gibt es schon eine Auswahl an Algorithmen (*Tahoe TCP, Reno TCP, Vegas TCP, Westwood TCP ...*[3]). Aber dieser muss meistens vor dem Start des Dienstes festgelegt werden. Als ein Beispiel für die Weiterentwicklung, wäre die Möglichkeit sinnvoll, im laufenden Betrieb den Algorithmus modifizieren oder austauschen zu können.

2.2 Frameworks

Betrachten wir ein paar Frameworks/Anwendungen. Eine Möglichkeit regelbasiert Pakete zu filtern, auszusortieren bietet iptables. Es ist ein Userspace-Programm um die Tabellen(tables), die durch die Firewall des Linux-Kernel bereit gestellt werden, bestehend aus einer Reihe von Netfilter-Modulen, zu konfigurieren [2]. Diese Tabellen enthalten Ketten und Regeln für die Behandlung von Paketen. Pakete werden durch sequentielles Abarbeiten von Regeln innerhalb einer Kette weitergereicht und es können auch Sprünge zwischen den Ketten erfolgen. Jede Regel spezifiziert auf welche Pakete sie zutrifft und kann auch ein Ziel (target), an die es weitergeleitet wird, enthalten (irgendein Dienst/Programm). Dies bietet Spielraum für Erweiterungen. Die Anwendung der Regeln können für Teile der Schichten(Layer) des OSI-Modells durchgeführt werden. Als Nachteil ist zu sehen, dass iptables auf IPv4 beschränkt ist und muss mit Root-Rechten ausgeführt werden. Für unterschiedliche Protokolle werden unterschiedliche Programme verwendet (ip6tables für IPv6, arptables für ARP, ebtables für Ethernet). Zudem ist iptables als Firewall konzipiert, also Pakete zu filtern, umzuleiten, auszusortieren.
Eine weitere Anwendung, die in Richtung Netzwerksicherheit geht, ist z.B. Snort [16][17]. Es ist ein Angriffserkennungs- und Angriffspräventionssystem (Network Intrusion Detection System, Network Intrusion Prevention System). Es kann außer zum Protokollieren von IP-Paketen, der Analyse des Datenverkehrs in IP-Netzwerken in Echtzeit, auch zum Erkennen von Angriffen und der unmittelbaren ereignisgesteuerten Blockierung eines solches Angriffes verwendet werden. Es liest direkt an der Netzwerkschicht den gesamten vorbeikommenden Netzwerk-Datenverkehr mit und vergleicht deren Inhalt mit charakteristischen Mustern von bekannten Angriffen. Signaturen werden diese Muster genannt und in Regeln(Rules) festgehalten. Zur Mustererkennung wird der Aho-Corasick-Algorithmus verwendet. Bei der Verarbeitung läuft ein Paket durch einen registrierten Satz von Präprozessoren, wobei jeder überprüft ob es ein für ihn relevantes Paket ist. Danach werden die Pakete durch eine Detections-Engine geschickt, die das Paket auf die zahlreichen Optionen checkt, die in der Snort Konfigurationsdatei stehen. Snort bietet die Möglichkeit der Erweiterbarkeit in Form von Plugins (Preprocessors - Vorverarbeitung, Detection Plugins - Verarbeitung, Output Plugins - Weitergabe), ist aber als solches ein sehr spezialisiertes

Programm.
In den anwendungsorientierten Schichten des OSI-Modells sind z.B. Webbrowser angesiedelt, die heutzutage über vielfältige Funktionen verfügen. Es geht von der Darstellung von Internetseiten (Darstellung medialer Inhalte verschiedenster Formate), Dateitransfer und -verwaltung (durch z.B. FTP), E-Mail, Chats bis zur Konfiguration von Endgeräten mit Webclient. Die meisten Browser bieten Erweiterungsmöglichkeiten in Form von Plugins oder Add-On's. Bei Firefox z.B. gibt es verschiedene Möglichkeiten. Eine Variante die Funktionalität von Firefox zu erweitern oder zu modifizieren sind Add-on's. Es gibt Add-on SDK Erweiterungen, die auf high-level JavaScript APIs aufbauen und wie die Restartless Erweiterungen keinen Browserneustart bei der Installation erfordern. Im Gegensatz dazu gibt es Overlay Erweiterungen, die einen Browserneustart erfordern. Diese Erweiterungen werden mit Hilfe von Standard Webtechnologien geschrieben, wie JavaScript, HTML, CSS und einiger bestimmter JavaScript APIs. Zudem gibt es neben Suchmaschinen-Plugins auch noch Plugins um die von Firefox nicht nativ unterstützten Inhalte anzuzeigen [11]. Firefox bietet daher verschiedene Ansatzpunkte für Erweiterungen ist aber auf die Anwendungsschicht beschränkt.
Ein weiteres Framework ist Click der modulare Router [10]. Click ist eine Architektur um flexible, konfigurierbare Router zu bauen. Ein Click Router setzt sich aus aneinander gehängten Ausführungsmodulen zusammen, die man Elemente nennt. Diese individuellen Elemente können z.B. einfache Routerfunktionen wie Paketklassifikation, Queuing, Scheduling und (interfacing) Interagieren mit Netzwerk-Schnittstellen (network devices) umsetzen. Eine Routerkonfiguration (router configurable) ist ein gerichteter Graph mit Elementen an den Knoten (vertices) und Pakete fließen entlang der Kanten des Graphs. Ein einfaches Beispiel ist in Abbildung 2.2 zu sehen. In Click kann man Protokolle, Dienste umsetzen, entwickeln, evaluieren von der Netzwerk-, über die Transport- bis zur Anwendungsschicht. Click bietet bereits zahlreiche Elemente für die verschiedensten Protokolle. Man braucht nicht notwendigerweise Root-Rechte und es gibt gute Anbindungen an Netzwerksimulatoren wie ns-3 [13], ns-2 [12], JiST/SWANS [9].

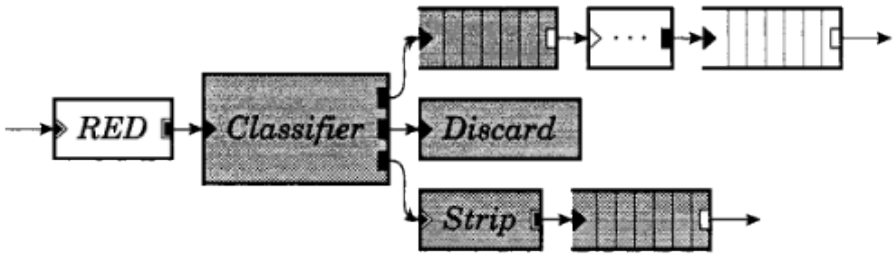

Abbildung 2.2: Flussbasierter Router-Kontext, ein Paket startet bei RED und stopt bei der ersten Queue die es trifft und durchläuft eines der drei Elemente. [10]

2.3 Nachladen von Programmcode

Wie könnten neue Funktionalitäten an ein bestehendes Programm angebunden werden.

- Quellcode (der Zielsprache)
- Quellcode (einer anderen Sprache)
- DSL (Domain Specific Language z.B. XML)
- Bytecode (Java, .Net, ..)

- Assembler (C, Basic, ..)
- Binary/Binärcode (C, C++, Pascal, ..)

Im Fall von Quellcode der Zielsprache muss dieser noch kompiliert werden, wobei je nach Sprache Binärcode oder Bytecode entsteht, der ins Programm eingebunden werden muss. Für manche Sprachen gibt es auch Interpreter, dort wird der Quellcode nicht kompiliert, wird aber bei jeder Ausführung erneut geparst und der Interpreter muss zusätzlich geladen werden.
Assembler ist sehr hardwarenah und lässt sich schnell übersetzen, verlangt aber vom Nutzer genaue Kenntnisse der CPU und ist wie Binärcode maschinenabhängig, ließe sich also nicht ohne Weiteres auf eine andere Architektur übertragen.
Quellcode einer anderen Sprache, als auch eine DSL müsste interpretiert werden oder mit einem entsprechenden Codegenerator in die Zielsprache übersetzt werden, worauf dann noch die selben Schritte wie beim direkten Einbinden des Quellcodes in der Zielsprache folgen müssten. Ein Beispiel dafür wäre ein per DTD oder XML-Schema validiertes XML-Vokabular, welches per XSLT in die Zielsprache transformiert werden könnte. Es ist aber aufgrund der vielen Übersetzungsschritte sehr zeitintensiv.
Im Fall von .Net, Java, etc. könnte man auch Bytecode direkt laden, dieser müsste aber vorher ebenfalls generiert oder durch Kompilieren erzeugt werden.
Viele Programmiersprachen bieten bereits Möglichkeiten Quellcode zur Laufzeit nachzuladen. Meistens wird die Funktionalität in Form einer Bibliothek, die man ins Projekt einbindet, zur Verfügung gestellt. Oder z.B. im Fall von Java sind bereits entsprechende Funktionalitäten in der Standard-API implementiert. Im Folgenden sind ein paar Beispiele für die Sprachen Java, C++ und C aufgeführt.
Bei **Java** gibt es mehrere Möglichkeiten. Wenn man ein JDK verwendet, kann man die Klassen im Package javax.tools nutzen um sich eine Java Compiler-Instanz zu generieren, die generierte Java-Klasse kompilieren und über einen ClassLoader laden und per Reflection darauf zugreifen (Ein Beispiel [8]). Alternativ kann man auch einen Bytecode-Generator [6][7] verwenden und direkt gültigen Java-Bytecode erstellen. Dafür muss man eine entsprechende API einbinden. Eine dritte Variante wäre, den generierten Quellcode als Java-Datei auf die Platte zu schreiben, per Systemaufruf zu kompilieren und das Ergebnis dann per ClassLoader zu laden und per Reflection auszuführen.
Im Falle von **C++** sind die Möglichkeiten nicht ganz so vielfältig. Eine Möglichkeit, um den generierten Quellcode auszuführen, wäre das Einbinden eines C++-Interpreters (wie folgender [5]), das heißt, der Code wird nicht in Binärcode übersetzt, sondern geparst und ausgeführt(interpretiert). Der Nachteil ist die verlangsamte Ausführungsgeschwindigkeit durch das bei der Ausführung immer wiederkehrende Parsen des Quellcodes und der Parser benötigt zusätzlichen Speicher.
Der allgemein gängigere Fall ist das dynamische Laden von vorkompilierten Shared Libraries (.so) oder Dll's (dynamic link library). Diese können natürlich auch per Aufruf eines externen Compilers zur Laufzeit erzeugt werden. Beispiele dafür sind das Einbinden von Plugins in ein bestehendes Programm (wie z.B. Winamp, Gimp, Wireshark ...), dabei wird ein Interface definiert, welches die Plugins implementieren müssen, welche dann zur Laufzeit als Shared Library oder Dll geladen werden.
Für **C** stellt der Tiny C Compiler eine weitere Möglichkeit zur Verfügung, mit dem man C-Quellcode zur Laufzeit kompilieren und ausführen kann. Dabei hat man die Wahl, ob man durch externen Aufruf eine Shared Library erzeugt und diese dann lädt oder der Code direkt im Speicher kompiliert und eingebunden wird. Er erfordert nicht viel Speicher, ist schnell und leicht einzubinden. (siehe [1])

Kapitel 3

Implementierung

3.1 Auswahl des Frameworks und der Programmiersprache

Da ich möglichst offen sein möchte in der Wahl des Netzwerklayers, des Protokolls in dem ich Funktionalitäten ersetzen, modifizieren oder neu einfügen möchte, habe ich mich für Click entschieden. Click lässt mir dahingehend alle Möglichkeiten offen und das Evaluieren wird durch die gute Unterstützung von Netzwerksimulatoren ebenfalls erleichtert.
Durch die Verwendung des Click-Frameworks bietet sich die Verwendung von C an. Click ist in C++ geschrieben von dem C eine Untermenge ist. Die Einschränkung auf C bedeutet aber nicht eine Einbuße an Funktionalität sonder lediglich an programmatischen Konstrukten, wie Klassen, Templates, Überladen von Operatoren, etc. .
Wenn man eine andere Programmiersprache wählen würde, wäre das Einbinden von Fremd-Code in die Click-Umgebung. Das ist immer mit zusätzlichen Aufwand verbunden (zeitlich, als auch beim Speicherbedarf), da ein entsprechender Interpreter oder Übersetzer eingebunden, geladen werden müsste. Betrachten wir z.B. Java, welches sich über JNI [14][4] einbinden ließe. Über diese Schnittstelle ist es möglich eine JVM zu laden und dann die entsprechende Klasse zu laden und auszuführen. Dies verlangt aber eine installierte Java-Version, verbraucht zusätzlich Speicher und Zeit, da zusätzlich eine JVM geladen werden muss. Des Weiteren müsste man noch eine Art Adapter implementieren der den JNI-Zugriff regelt.
Diesen Overhead kann ich mir sparen, wenn ich als Sprache C wähle. C ist auch schneller als Java und bietet komfortable Möglichkeiten auf Byte-/Bit-Ebene zu operieren, um z.B. Headerinformationen eines Netzwerkpaketes effizient zu extrahieren. Zudem bietet für C der Tiny C Compiler [1] einen kleinen und schnellen C-Compiler, der das Kompilieren und Einbinden von C-Code in ein laufendes C/C++ Programm ermöglicht. Er ist deutlich kleiner als der gcc und laut Herausgeber auch um ein Vielfaches schneller beim Kompilieren von C-Code. Er kann C-Code direkt (im Speicher) kompilieren und ausführen, es ist kein Linken oder Assemblieren nötig und ein voller C-Präprozessor und GNU-Like Assembler sind inkludiert. Man muss lediglich die libtcc einbinden und kann so (dynamisch generierten/übersetzten) Code zur Laufzeit einbinden.
Der TCC ermöglicht es Dateien mit C-Code oder dynamisch generierten C-Code, der eine oder mehrere Funkionen enthalten kann, welche sich auch gegenseitig aufrufen können, ins Programm zu laden und sich Pointer auf die entsprechenden Funktionen zu holen, über welche man diese aufrufen kann. Wir beschränken uns daher auf das Ersetzen/Überschreiben von Funktionen.

3.2 Umsetzung

Um die Funktionalitäten des TCC, wie im letzten Abschnitt beschrieben, nutzen zu können, muss die C-Bibliothek libtcc eingebunden werden. Dabei ist es bei meiner Implementierung egal, ob der C-Code dynamisch generiert wurde, aus einer C-Datei geladen wird oder von einer anderen Quelle kommt. Der C-Code-String wird geparst, um die Signaturen der enthaltenen Funktionen zu extrahieren. Diese umfassen den Namen der Funktion, den Rückgabetyp und die Anzahl, Reihenfolge und Typen der Parameter, sofern vorhanden. In C kann man nur typisierte Funktionspointer verwenden, also zur Kompilierungszeitpunkt müssen Rückgabe und Parametertypen bekannt sein. Wie im folgenden Beispiel zu sehen:

Listing 3.1: Beispiel Funktions-Pointer

```
char * read_line_from_file(FILE* my_stdin) {
  ...//originale Funktion
}
//Verwendung des Pointers auf die Funktion
int main() {
  char* s;
  char* (*readLine)(FILE*) = read_line_from_file;
  s = readLine(stdin);
  ...
}
```

Um dies zu umgehen wird nicht direkt auf die Funktionen zugegriffen, sondern der Umweg über eine Wrapper-Funktion gegangen, um den Zugriff zu generalisieren. Die Wrapper-Funktion ruft dann die eigentliche Funktion auf. Alle Wrapper haben bis auf ihren Namen identische Signaturen. Als Übergabeparameter haben sie ein Array vom Typ eines speziell definierten Union und geben ein solches Union auch wieder zurück.

Listing 3.2: Union mit unterstützten C-Typen

```
/** Erfasst die Uebergabe-/Rueckgabewerte. */
typedef union {
    int i;                  //Integer
    int* i_p;               //Pointer auf Integer
    unsigned int u_i;       //positiver Integer
    unsigned int* u_i_p;    //Pointer auf positiven Integer
    short s;                //Short
    short* s_p;             //Pointer auf Short
    unsigned short u_s;     //positiver Short
    unsigned short* u_s_p;  //Pointer auf positiven Short
    long l;                 //Long
    long* l_p;              //Pointer auf Long
    unsigned long u_l;      //positiver Long
    unsigned long* u_l_p;   //Pointer auf positiven Long
    float f;                //Float
    float* f_p;             //Pointer auf Float
    double d;               //Double
    double* d_p;            //Pointer auf Double
    long double ld;         //Long Double
    long double* ld_p;      //Pointer auf Long Double
    char c;                 //Character
    char* c_p;              //Zeichenkette (String)
    void* p;                //unbestimmter Pointer
} t_value;
```

Dieses Union umfasst alle (Daten-)Typen, die ich unterstützen will. Es kann immer nur den Wert eines Typs annehmen und hat den Vorteil, dass es eine feste Länge besitzt (Byte-Länge des größten unterstützten Typs). Jedes Union (Konzept C Struct/Union [19]) aus dem Übergabe-Array muss also einen

Wert entsprechend des Übergabeparameters der originalen Funktion zugewiesen bekommen und das Array muss die Länge der Anzahl der Parameter der originalen Funktion haben. Der Wrapper weist beim eigentlichen Funktionsaufruf die Werte der Unions den entsprechenden Parametern zu. Der Rückgabewert der Funktion wird ebenfalls in ein solches Union abgelegt, welches vom Wrapper zurückgegeben wird.
Die Wrapper werden mit Hilfe der extrahierten Header-Informationen generiert und mit an den originalen C-Quellcode-String angefügt, bevor der Quellcode übersetzt und eingebunden wird. Im Programm werden nur die Pointer auf die Wrapper verwendet.

Listing 3.3: zwei Beispielfunktionen und deren generierte Wrapper

```
//originale Funktionen
double sum(double d1, double d2){
        return d1 + d2;
}

void echo(char* s) {
        printf("%s\n",s);
}
//generierte Wrapper
//erwartet ein t_value Array der Laenge zwei
t_value f_wrapper_sum(t_value* param) {
        t_value result;
        //Uebergabe der Parameter an die eingentliche Funktion
        result.d = sum(param[0].d, param[1].d);
        return result;
}
//erwartet ein t_value Array der Laenge eins
t_value f_wrapper_echo(t_value* param) {
        t_value result;
        result.p = NULL;
        echo(param[0].c_p);
        return result;
}
```

Welche Einschränkungen ergeben sich? Der dynamisch nachladbare C-Quellcode ist auf den vom Tiny C Compiler unterstützten Sprachumfang beschränkt. Das beinhaltet den ANSI-C Standard, einschließlich der vollen Unterstützung der Typen long double, double und float. Zudem sind viele Features des ISO C99 Standards umgesetzt (z.B. volle Unterstützung der Typen long long und _Bool, ...). Einige GNU-C Erweiterungen sind ebenfalls implementiert [1].
Man ist auf die Sprachkonstrukte von C beschränkt, also spezielle C++, Konstrukte (wie z.B. Klassen, Templates etc.) können nicht verwendet werden.
Folgende Einschränkungen habe ich festgelegt: Die Übergabeparameter der Funktionen sind auf die Standard-C Typen und Pointer auf diese (Arrays) begrenzt. Die komplexen Datentypen wie Structs oder Unions sind nicht zugelassen. Diese wären in der Anwendung nicht bekannt und um diese zu unterstützen müsste man eine Art Reflection (wie in Java) implementieren. Man kann die Elemente eines Structs aber auch auf die Parameter der Funktion verteilen, etc.
Innerhalb der Funktionen ist nur die Verwendung von Funktionen der Standardbibliotheken oder selbstdefinierter Funktionen (müssen sich im selben Quellcode-String befinden) zugelassen, da man sonst den C-Code nach Funktionen scannen, die passende Bibliothek finden und laden müsste, oder im Fall der Nichtzuordnungsbarkeit, den Code ablehnen müsste.
Bis auf per define-Anweisung definierte Konstanten, sind keine externen Variablen innerhalb des Quellcodes zugelassen.
Die Funktionen sind in der Anzahl ihrer Parameter nicht eingeschränkt und können die von mir zugelassenen Standardtypen annehmen [3.2]. Eine Variable Anzahl an Parametern, wie z.B. bei printf(..), ist allerdings nicht zugelassen.

Es sind auch Pointer auf die Typen zugelassen, also Arrays als Übergabeparameter. In diesen Zusammenhang habe ich noch für jeden Pointer-Typen einen zusätzlichen Variablentypen definiert, denn man jeweils als Return-Typ der Funktion angeben kann.

Listing 3.4: Pointer Typen

```
typedef int* _ref_int_;
typedef unsigned int* _ref__unsigned_int_;
typedef short* _ref__short_;
typedef unsigned short* ref__unsigned_short_;
typedef long* _ref__long_;
typedef unsigned long* _ref__unsigned_long_;
typedef float* _ref__float_;
typedef double* _ref__double_;
typedef long double* _ref__long_double_;
typedef char* _ref__char_;
```

Die Verwendung eines solchen Rückgabetypen gibt an, dass der Rückgabewerte den Inhalt eines Übergabeparameters referenziert. Das spielt eine Rolle beim automatischen Freigeben der referenzierten Übergabeparameter und Rückgabewerte. Dann wird der Rückgabewert nicht automatisch freigegeben. Wenn der Rückgabewert und ein Übergabeparameter auf denselben Speicherbereich zeigen, dann würde versucht werden, diesen zweimal freizugeben, was nicht erlaubt ist und zum Programmabsturz führen kann.

3.3 Einbindung in Click

Das von mir Geschriebene wird als Click-Element eingebunden. Das Element besitzt einen Funktionspointer der Form:

Listing 3.5: Funktionszeiger des TCC-Elementes

```
void* (*simpleAction) (void* packet) = NULL;
```

Dieser ist initial NULL. Über einen Handler ([10]) kann man C-Quellcode als Zeichenkette an das Element schicken. Dieser Quellcode wird dann kompiliert und eingebunden. Der gesendete Quellcode muss aus einer Funktion bestehen, die dem Funktionszeiger des TCC-Elementes entspricht und einen festen Namen besitzt, z.B. 'simpleAction'. Bei erfolgreicher Kompilierung wird diese Funktion dann dem Zeiger des TCC-Elementes zugewiesen.
Solange der Zeiger NULL ist, werden die ankommenden Pakete einfach durchgereicht, ansonsten werden sie der Funktion übergeben und der Rückgabe Wert weitergereicht.

Kapitel 4

Evaluation

Da als Einsatzfeld in Click Paketfilter in Frage kommen, gilt es zu Evaluieren, wie lange das Kompilieren und Einbinden des Quellcodes durchschnittlich benötigt und wovon diese Zeit abhängt. Daraus kann man dann Rückschlüsse ziehen, bis zu welcher Paketfrequenz und bei welchem Quellcodeumfang das Kompilieren und Einbinden noch ohne Paketverluste stattfinden kann.
Denkbar sind unterschiedlich komplexe Paketfilter, also die Zeit zu messen für simple versus komplexe Filter oder auch kurze versus lange Funktionskörper. Des Weiteren könnte die Zeit von der Strukturiertheit des Quellcodes abhängen. Z.B. kann man bei einer komplexen Funktion, Funktionalität in andere Funktionen auslagern, die dann aufgerufen werden. Also betrachten wir wenige Funktionen (wenig Strukturiertheit) versus viele Funktionen (viel Strukturiertheit). Eine weitere Form der Komplexität und Strukturiertheit einer Funktion ergibt sich aus der Anzahl ihrer Parameter (wenig Parameter gleich geringere Komplexität und mehr Struktur, versus vielen Parametern gleich höhere Komplexität und weniger Struktur). Daraus ergeben sich hauptsächlich drei zu betrachtende Szenarien:

1. Eine Funktion mit wachsender Parameterzahl. (Listings 4.1,4.2)
2. Steigende Anzahl an Funktionen. (Listing 4.3)
3. Eine Funktion mit wachsendem Funktionskörper. (Listing 4.4)

Um in Fall 1 mögliche Kompileroptimierungen bezüglich nicht verwendeter Funktionsparameter innerhalb einer Funktion abzudecken, wird noch die Messung mit leerem Funktionskörper (siehe Listing 4.1) und mit minimalem Funktionskörper (siehe Listing 4.2) unterschieden. Beim minimalem Funktionskörper werden alle Parameter arithmetisch miteinander verknüpft, einer Ergebnisvariablen vom Rückgabetyp der Funktion zu gewiesen und diese dann zurückgegeben.
Beispielquellcode für Fall 2 und 3 ist in den Listings 4.1 und 4.2 zu sehen. In diesen Fällen werden auch alle Funktionsparameter im Funktionskörper verwendet.
Alle Funktionen sind per Quellcodegenerator generiert. Die Typen der Parameter und Rückgabewerte der generierten Funktionen, als auch die Anordnung der verwendeten Sprachkonstrukte (if-then-else, for, do, while ...) ergeben sich dabei durch einen Zufallsgenerator.
Für jeden Durchschnittswert werden 1000 Kompilierungs- und Einbindungsdurchläufe für denselben Quellcode am Stück ausgeführt, durch die Anzahl der Ausführungen geteilt, dieses wird 100 mal wiederholt und daraus der Mittelwert bestimmt. Es wird nur die Zeit betrachtet, die das Kompilieren und Einbinden in Anspruch nimmt. Die Ausführungszeit der jeweils eingebunden Funktionen wird nicht betrachtet.
Die Messungen wurden auf einem Rechner mit 1TB Arbeitsspeicher und mit 4 Intel(R) Xeon(R) E7-8880 v2 2.50GHz (maximale Turbo-Frequenz 3.10GHz) Prozessoren durchgeführt. Jeder Prozessor hat 37.5MB Cache, 15 Kerne und 30 Threads.

Aufgrund der Tatsache, dass eine Vorverarbeitung des Quellcodes (Einlesen des Quellcodes, Extrahieren der Funktionsheader, Erzeugen der Wrapper, Kompilieren, Anlegen der Speicherstrukturen) stattfindet, ist in allen drei Fällen (Zunahme der Parameter, der Funktionsanzahl oder des Funktionskörpers) zu erwarten, dass die benötigte Zeit ansteigt.

Listing 4.1: Beispiel Messung 1.1

```
char func_cabdefgh() {
    return 'k';
}
unsigned int* func_97616621(float* f_p__7376) {
    return NULL;
}
long double* func_61645717(unsigned short u_s__4778, long* l_p__5005) {
    return NULL;
}
unsigned short func_61961289(int* u_i_p2051, short* s_p__2155, long* l_p__6616) {
    return 0;
}
...
```

Listing 4.2: Beispiel Messung 1.2

```
char func_ijklmnop() {
    static char res;
    res = 'l';
    return res;
}
...
void* func_12274884(unsigned long* u_l_p9494, float f____3989, unsigned long* u_l_p7886)
    {
    static long res;
    res = (long )(*u_l_p9494 *  -f____3989 *  --(*u_l_p7886));
    return &res;
}
...
```

Listing 4.3: Beispiel Messung 2

```
float* func_63190628(char c____5251, char c____9080, double d____6505) {
    static float res;
    res = (float )(c____5251 + c____9080 - d____6505);
    return &res;
}
int func_79802836(unsigned long u_l__5044, unsigned long* u_l_p2163, float f____7823) {
    static int res;
    res = (int )(u_l__5044 * *u_l_p2163 * f____7823);
    return res;
}
...
```

Listing 4.4: Beispiel Messung 3

```
long double* func_27834889(char c____2932, unsigned short u_s__1757, unsigned short*
    u_s_p4420) {
    static long double res;
    res = (long double )(c____2932 / u_s__1757 +  ++(*u_s_p4420));
    return &res;
}
unsigned int func_99083273(char* c_p__9109, unsigned long u_l__8533, unsigned long*
    u_l_p9027) {
```

```
    static unsigned int res;
    while(*u_l_p9027 > (unsigned long )-2147115039) {
        unsigned int* u_i_p9541 = malloc(sizeof(unsigned int ));
        *u_i_p9541 = (unsigned int )862313;
        *u_l_p9027 = (unsigned long )(*((long*)c_p__9109) - *u_l_p9027);
        while(*((long*)c_p__9109) <= (char )*u_l_p9027 && u_l__8533 != (unsigned long )*((
            long*)c_p__9109)) {
            *u_l_p9027 = (unsigned long )(*((long*)c_p__9109));
            *c_p__9109 /= (char )(*u_i_p9541);
        }
        free(u_i_p9541);
    }
    res = (unsigned int )(*((long*)c_p__9109) / u_l__8533--  * (*u_l_p9027)-- );
    return res;
}
...
```

4.1 Zunehmende Parameterzahl mit leerem Funktionskörper

Es wird für 21 Funktionen mit zunehmender Parameterzahl (0 bis 20 Parameter) und leerem Funktionskörper gemessen (Listing 4.1). Pro Funktion werden 100 mal 1000 Kompilier- und Einbindungsvorgänge ausgeführt und der zeitliche Mittelwert gebildet.
Es ist zu erwarten, dass mit zunehmender Parameterzahl auch die benötigte Zeit zunimmt. Da die Vorverarbeitung aufwendiger wird und auch die Quellcodelänge größer wird.

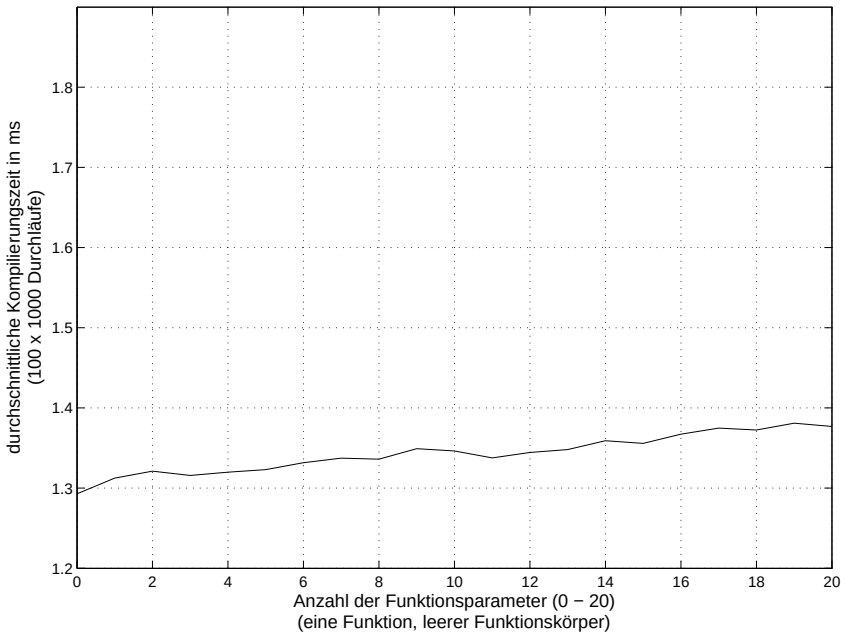

Abbildung 4.1: Durchschnittliche Kompilierungszeit für 100 mal 1000 Durchläufe bei einer Funktion mit zunehmender Parameterzahl und leerem Funktionskörper.

Wie in Abbildung 4.1 zu sehen, steigt der zeitliche Verbrauch mit zunehmender Parameterzahl an. Die Kompilierzeit wächst von 1.293ms bis auf 1.381ms an. Es fallen zeitliche Schwankungen auf. Z.B. braucht

die Messung mit 11 Funktionsparameter weniger lang als die mit 10 Parametern. Diese Schwankungen könnten durch die Beeinflussung durch parallel laufende Prozesse auftreten.

4.2 Zunehmende Parameterzahl mit minimalem Funktionskörper

Es wird für 21 Funktionen mit zunehmender Parameterzahl (0 bis 20 Parameter) und minimalem Funktionskörper gemessen (Listing 4.2). Pro Funktion werden 100 mal 1000 Kompilier- und Einbindungsvorgänge ausgeführt und der zeitliche Mittelwert gebildet.
Es ist zu erwarten, dass mit zunehmender Parameterzahl auch die benötigte Zeit zunimmt. Da die Vorverarbeitung aufwendiger wird und auch die Quellcodelänge größer wird.

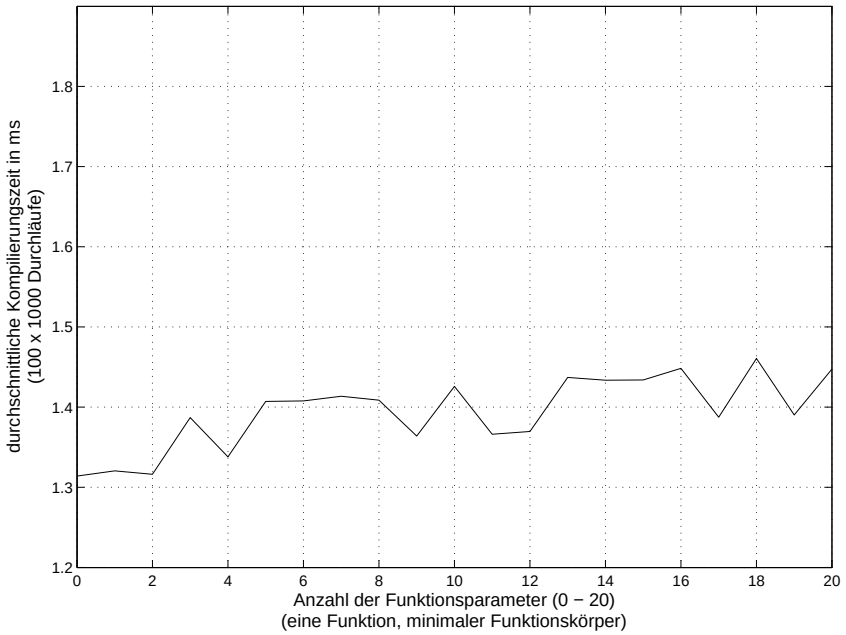

Abbildung 4.2: Durchschnittliche Kompilierungszeit für 100 mal 1000 Durchläufe bei einer Funktion mit zunehmender Parameterzahl und minimalem Funktionskörper.

Wie in Abbildung 4.2 zu sehen, steigt der zeitliche Verbrauch mit zunehmender Parameterzahl an. Die Kompilierzeit wächst von 1.3141ms bis auf 1.4607ms an. Es sind ebenfalls Schwankungen wie bei der vorhergehenden Messreihe zu sehen, diese fallen deutlich stärker aus.

4.3 Zunehmende Anzahl an Funktionen

Bei dieser Messung gibt es 20 generierte Funktionen mit minimalem Funktionskörper und drei zufälligen Parametertypen und einem zufälligem Rückgabetyp (Listing 4.3). Die Messung geht vom Kompilieren und Einbinden von einer bis zwanzig Funktion. Pro Funktionssatz werden 100 mal 1000 Kompilier- und Einbindungsvorgänge ausgeführt und der zeitliche Mittelwert gebildet.
Es ist zu erwarten, dass mit steigender Funktionsanzahl auch die benötigte Zeit ansteigt, durch die aufwendiger werdende Vorverarbeitung bei zunehmender Funktionszahl und dem länger werdenden Quellcode. Die zeitliche Zunahme sollte deutlicher ausfallen, als bei den beiden vorhergehenden Messreihen, da der

Quellcodezuwachs und der Vorverarbeitungsaufwand pro zusätzliche Funktion größer ist, als wenn sich lediglich die Parameterliste einer Funktion verlängert.

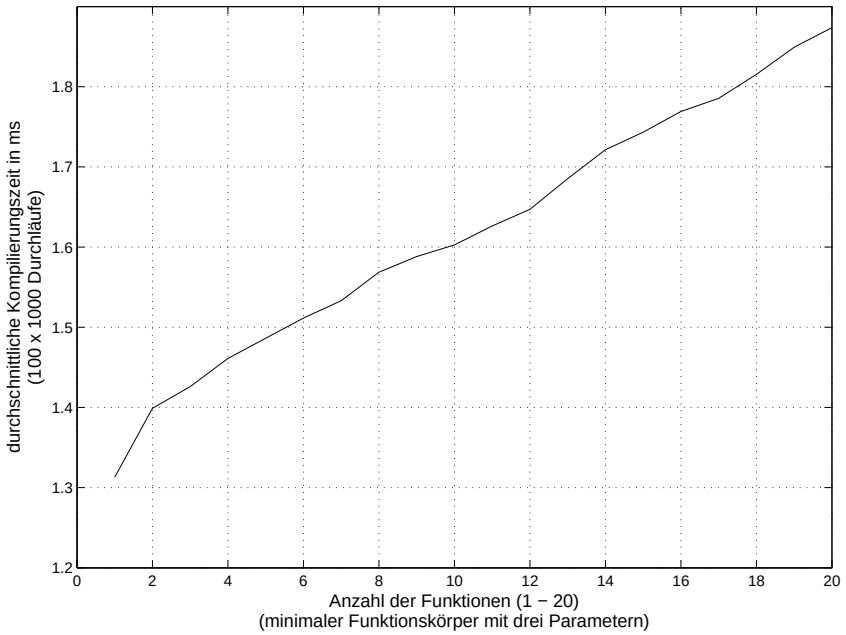

Abbildung 4.3: Durchschnittliche Kompilierungszeit für 100 mal 1000 Durchläufe bei zunehmender Funktionsanzahl mit jeweils drei Parametern und minimalem Funktionskörper.

Wie in Abbildung 4.3 zu sehen, steigt der zeitliche Verbrauch mit zunehmender Funktionsanzahl an. Die Kompilierzeit wächst von 1.313ms bei einer Funktion bis auf 1.8737ms bei 20 Funktionen an. Die zeitliche Zunahme ist deutlicher als bei den beiden vorhergehenden Messreihen.

4.4 Zunehmende Länge des Funktionskörpers

Bei dieser Messung gibt es 20 generierte Funktionen mit anwachsendem Funktionskörper und drei zufälligen Parametertypen und einem zufälligem Rückgabetyp (Listing 4.4). Die Konstrukte innerhalb der Funktionskörper wurden ebenfalls vom Quellcodegenerator per Zufallsgenerator zusammengesetzt. Die Länge wird in Bytes angegeben und geht von 209 - 8900Bytes. Der Byte-Zuwachs von einer Messung zur darauffolgenden ist nicht äquidistant (resultiert aus der Generierung per Zufallsgenerator). Pro Funktion werden 100 mal 1000 Kompilier- und Einbindungsvorgänge ausgeführt und der zeitliche Mittelwert gebildet.
Es ist zu erwarten, dass mit steigender Größe des Funktionskörpers auch die benötigte Zeit ansteigt. Denn denn der Aufwand für das Parsen und Kompilieren sollte zunehmen.

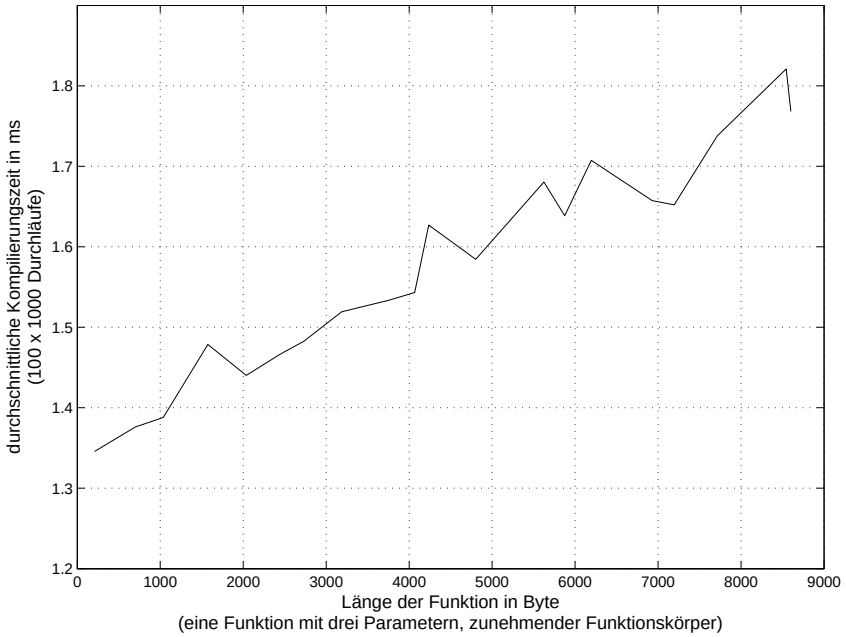

Abbildung 4.4: Durchschnittliche Kompilierungszeit für 100 mal 1000 Durchläufe bei einer Funktion mit drei Parametern und zunehmendem Funktionskörper.

Wie in Abbildung 4.4 zu sehen, steigt der zeitliche Verbrauch mit anwachsendem Funktionskörper an. Die Kompilierzeit wächst von 1.3457ms bis auf 1.8211ms an. Es liegen ebenfalls zeitliche Schwankungen bei der Messreihe vor.

4.5 Ergebnis

Wie erwartet, steigt in allen Messungen der Zeitverbrauch mit zunehmender Parameter-, Funktionszahl oder Funktionskörperlänge an. Eine zunehmende Parameter- als auch Funktionsanzahl impliziert ebenfalls eine ansteigende Byte-Anzahl des zu kompilierenden Quellcodes. Daraus könnte sich ergeben, dass die Zunahme des Zeitverbrauchs eher auf die Länge des Quellcodes und nicht auf die Parameter- oder Funktionsanzahl zurückzuführen ist. Dies könnte Bestandteil weiterer Messungen sein.
Bis auf die Messreihe mit zunehmender Funktionsanzahl treten bei allen Messungen zeitliche Schwankungen auf. Wie schon erwähnt, könnten die Schwankungen durch die Beeinflussung durch parallel laufende Prozesse auftreten. Die Schwankungen verschwanden aber auch nicht durch Erhöhung der Messanzahl der Kompilier-/Einbindevorgänge von ursprünglich 1000 pro Quellcodekonstellation auf 100 mal 1000 Messungen.
Für die hier betrachtete Hardwarekonfiguration, war bei geringer Funktionsanzahl, Parameterlänge, als auch kurzem Funktionskörper eine Durchschnittszeit von 1.293 Millisekunden möglich. Damit wäre bei einer Paketfrequenz von ca. 0,773 Hz eine Kompilierung/Einbindung für jedes Paket möglich ohne den Paketfluss zu stören. Dabei ist die Zeit für die Ausführung der Funktion nicht berücksichtigt. Wie an den Messreihen zu sehen, steigt der Zeitverbrauchswert schnell an, was zu zu großer Verzögerung und Verlusten von Paketen führen kann. Das Nachladen der Funktionalität im laufenden Betrieb ist in der jetzigen Form und für die betrachtete Konfiguration nicht praktikabel für einen hochfrequenten Einsatz.

Kapitel 5

Zusammenfassung und Ausblick

Nach der Abwägung der Möglichkeiten Quellcode nachzuladen und die Betrachtung gegebener Netzwerkanwendungen/-Frameworks, viel die Wahl auf C als Programmiersprache, der Tiny-C Compiler als Hilfsmittel zum dynamischen nachladen von C-Quellcode und die Verwendung des Netzwerkframeworks Click.
Es wurde erfolgreich unter Verwendung des Tiny-C Compilers eine Variante zum dynamischen Nachladen von C-Quellcode in ein laufendes C/C++ Programm implementiert. Die Implementierung wurde erfolgreich als Element in dem Netzwerkframework Click umgesetzt.
Aus der Evaluierung folgt, dass mit zunehmender Komplexität oder Strukturiertheit des Quellcodes der zeitliche Aufwand für das Kompilieren und Einbinden des Quellcodes ansteigt. Bei der für die Messungen verwendeten Hardwarekonfiguration, ergab sich, für die umgesetzte Implementierung, dass eine hochfrequente Anwendung (z.B. bei einem Paketfilter für jedes Paket Quellcode kompilieren und einbinden) nicht praktikabel ist. Für einfache Filter (geringe Komplexität und Struktur des Quellcodes) ergab sich eine maximale Paketfrequenz von ca. 0,773 Hz, bei der eine pro Paket Ausführung noch ohne Paketverluste möglich wäre.
Bei den Messungen kam es zu relativ großen Schwankungen im zeitlichen Verbrauch für die Kompilier- und Einbindungsvorgänge, was eine qualitative Abschätzung des zeitlichen Aufwandes in einer realen Anwendung erschwert.
Die Implementierung ist mit unter 300KB sehr klein und leicht in jedes C/C++-Programm einzubinden, welches ein breites Einsatzfeld ermöglicht. Mit Click hat man zudem ein umfangreiches Framework zum Erstellen/Simulieren von Netzwerkanwendungen. Als Ausblick wäre eine Anwendung im Bereich von Sensornetzwerken denkbar, z.B. um die Auswertungsmethoden im Sensornetzwerk für Sensordaten im laufendem Betrieb auszutauschen oder zu erweitern.

Abbildungsverzeichnis

2.1 OSI-Schichtenmodell 2
2.2 Einfaches Beispiel: flussbasierter Router-Kontext 4

4.1 Messung 1.1: Durchschnittliche Kompilierzeit bei einer Funktion mit zunehmender Parameterzahl und leerem Funktionskörper 12
4.2 Messung 1.2: Durchschnittliche Kompilierzeit bei einer Funktion mit zunehmender Parameterzahl und minimalem Funktionskörper 13
4.3 Messung 2: Durchschnittliche Kompilierzeit bei zunehmender Funktionsanzahl mit fester Parameterzahl und minimalem Funktionskörper 14
4.4 Messung 3: Durchschnittliche Kompilierzeit bei einer Funktion mit fester Parameterzahl und zunehmendem Funktionskörper 15

Literaturverzeichnis

[1] Bellard, Fabrice: *Tiny C Compiler*. https://bellard.org/tcc/,

[2] *iptables*. https://de.wikipedia.org/wiki/Iptables,

[3] Grieco, Luigi A. ; Mascolo, Saverio: Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control. In: *SIGCOMM Comput. Commun. Rev.* 34 (2004), April, Nr. 2, 25–38. http://dx.doi.org/10.1145/997150.997155. – DOI 10.1145/997150.997155. – ISSN 0146–4833

[4] IBM: *Java Invocation API*. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzaha/invocapi.htm,

[5] *C/C++ Interpreter*. http://www.softintegration.com/,

[6] *ASM - all purpose Java bytecode manipulation and analysis framework*. https://asm.ow2.io/,

[7] *Java bytecode engineering toolkit since 1999*. http://www.javassist.org/,

[8] *Java Code zur Laufzeit generieren (Beispiel)*. https://www.java-forum.org/thema/code-zur-laufzeit-generieren.61076/,

[9] *JiST / SWANS*. http://jist.ece.cornell.edu/,

[10] Kohler, Eddie ; Morris, Robert ; Chen, Benjie ; Jannotti, John ; Kaashoek, M. F.: 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

[11] *Ändern und Erweitern von Mozilla-Anwendungen*. https://developer.mozilla.org/de/docs/Mozilla/Add-ons,

[12] *Ns-2 is a discrete event simulator targeted at networking research*. http://nsnam.sourceforge.net/wiki/index.php/Main_Page,

[13] *ns-3 is a discrete-event network simulator for Internet systems*. https://www.nsnam.org/,

[14] Oracle: *The Invocation API*. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/invocation.html,

[15] *OSI-Modell*. https://de.wikipedia.org/wiki/OSI-Modell, September 2007

[16] *Snort*. https://de.wikipedia.org/wiki/Snort,

[17] *SNORT Users Manual 2.9.11*. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/snort_manual.html,

[18] *Transmission Control Protocol (TCP)*. https://de.wikipedia.org/wiki/Internetwork_Packet_Exchange,

[19] Verlag, Rheinwerk: *Konzept C Struct/Union*. http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/015_c_strukturen_009.htm,