Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Bericht

Seminar Concurrent Programming 2013

12. Juni 2013

Mesh-Network in C mit Pthreads

Autor
Daniel Ritz
Studiengang Informatik, i09b

Betreuer
Tomáš Pospíšek

Inhaltsverzeichnis

Zusammenfassung

Im Rahmen des Seminar „Concurrent Programming“ an der ZHAW wurde erfolgreich ein einfacher Mesh Switch entwickelt. Dieser musste in C unter der Verwendung von Pthreads implementiert werden. Ein einfaches Protokoll wurde in der Klasse definiert und festgehalten. Daraus ergab sich auch die Anforderung, dass die Implementationen der verschiedenen Studenten untereinander funktionieren müssen.

Der Mesh Switch hat die Aufgabe Pakete durch ein Mesh-Netzwerk zu routen um so von einem Anfangspunkt zu einem Endpunkt zu gelangen. Damit dies gelingen kann, wird anfänglich jeder Switch das Netzwerk fluten. Der Empfänger einer Nachricht bestätigt dem Sender den Erhalt. Daraus müssen sich die einzelnen Switche eine Route „berechnen“.

Die Implementation beruht auf reinem C, ohne zusätzliche Libraries. Die Verwendung von Threads erlaubt einfache blockierende Ein-/Ausgabe. Um mit verschiedenen Funktionen der Pthreads-Library zu experimentieren, aber auch um verschieden Datenstrukturen auszuprobieren, wurden neben einer blockierenden Queue auch eine Kombination aus einem Ring-Buffer und einer Hashtabelle implementiert.

Insgesamt war es eine interessante Aufgabe und eine gute Übung für den Umgang mit Pthreads, aber auch für die Verwendung von BSD Sockets.

Kapitel 1

Einführung

1.1 Aufgabenstellung

Nachfolgend die Aufgabenstellung so wie sie im Wiki (http://edu.panter.ch/Mesh) vorgegeben ist.

Übersicht

Das Ziel ist mithilfe von Pthreads einen sehr simplen Mesh Paket Switch zu machen. Die Mesh Switche fluten das Netzwerk um eine Route zu finden.

Anforderungen

- kein global Lock
- Verwendung von Pthreads
- muss vorgegebenes Testprogramm bestehen
- muss interoperabel mit Mesh-Switchen der anderen Studenten sein
- Das Projekt soll im „tgz“ Format abgegeben werden
- Das Projekt muss per „make“ gebaut werden können

Spezifikation

Paketformat

- Zahlen in Network Byte Order
- Paketstruktur

2 Bytes	1 Byte	1 Byte	128 Bytes
Paket-ID	Ziel (1) oder Quelle (0)	Paket Type ('C','O','N')	Inhalt

Tabelle 1.1: *Paketformat*

Paket-Protokoll

- wenn Paket Typ == 'C', dann ist es ein Paket welchen Inhalt übermittelt
  - der Inhalt des Pakets enthält dann die zu übermittelnde Fracht
  - in Mesh Switch, leitet nur das erste Paket mit einer Paket-ID weiter, die folgenden Pakete mit der gleichen Paket-ID wirft er weg
- wenn Paket Typ == 'O', dann ist es ein 'OK' Paket, das signalisiert, dass das Paket angekommen ist, dabei ist die Paket-ID identisch mit jener des 'C' Pakets, welche bestätigt wird
  - ein Paket das das Ziel (oder die Quelle) erreicht wird mit einem Typ 'O' Paket mit identischer Paket-ID beantwortet
  - wenn nach einer Konfigurierbaren Zeitdauer kein 'O' Paket kommt, dann gilt die Route, welche das Paket genommen hat als nicht zum Ziel führend
- wenn Paket Typ == 'N', dann soll sich der Mesh Switch mit einem neuen Nachbar verbinden
  - der Inhalt des Pakets enthält dann 4 Bytes mit der IP Adresse und 2 Bytes mit dem TCP Port des Nachbarn

Start des Executables

```
mesh.exe <portnum> [-q|-z]
```

- *portnum* ist eine TCP Portnummer - default ist 3333
- es kann entweder -q oder -z oder nicht als zusätzliches Argument angegeben werden
- wenn als Argument *-q* angegeben wird, dann ist der Mesh Router die Quelle
- wenn als Argument *-z* angegeben wird, dann ist der Mesh Router das Ziel

Zur Verfügung gestellter Code

- TCP/IP Client/Server
  - Auf https://github.com/tpo/common_network steht ein Demo Server/ Client zur Verfügung
  - er kann Kommandozeilen auswerten
- Test Programm
  - kann testen, ob das Mesh funktioniert
  - https://github.com/tpo/mesh-network-py

Optional, interessant

IP Tunneln über Mesh Switch Netz via TUN-Interface http://vtun.sourceforge.net/tun/faq.html

Kapitel 2

Design

2.1 Grundsätzliche Überlegungen

Die Einfachheit des zu implementierenden Protokolls würde anbieten, die Applikation ohne Pthreads zu implementieren. Stattdessen könnte die Applikation auf nicht-blockierende Ein-/Ausgabe setzen und mittels einer Ereignis-Schlaufe („Event Loop“ in verständlicher Sprache) mit `poll()`, `select()`, oder je nach Plattform mit „ePoll“ oder „kqueues“, die Daten verarbeiten. Da die Aufgabenstellung aber zwingend die Verwendung von Pthreads vorgibt, wird mit blockierender Ein-/Ausgabe und einem Thread für die lesende Seite jeder Verbindung verwendet.

Im Kern steht pro Verbindung ein Thread der blockierend Pakete empfängt. Diese werden ausgewertet und, falls sie weiter gesendet werden müssen, in eine Queue gelegt. Weitere Worker-Threads entnehmen der Queue die Pakete wieder und senden diese weiter.

2.2 Architektur und Komponenten

Die Applikation, nachfolgend „*Meshy*“ genannt, ist wie in Abbildung 2.1 gezeigt aufgebaut. Die einzelnen Komponenten werden hier kurz beschrieben.

2.2.1 Connections

Die „Connections“ sind eine Liste aller aktiver TCP/IP-Verbindungen. Diese Liste ist dynamisch, da jederzeit ein anderer Knoten eine Verbindung aufbauen kann (`accept()`) und da mittels „N“-Paket eine Verbindung zu einem Nachbarn aufgebaut werden kann (`connect()`).

2.2.2 Packet ID Cache

Der „Packet ID Cache“ merkt sich Paket-IDs in Verbindung mit der Adresse (1 für Ziel, 0 für Quelle) die vom aktuellen Knoten bereits einmal empfangen wurden. Die Zwischenspeicherung dieser IDs zusammen mit einigen zusätzlichen Information ist unerlässlich aus zwei Gründen:

- Pakete dessen ID bereits einmal empfangen wurden, müssen verworfen werden.
- Wird ein O-Paket empfangen, muss anhand der ID und der Adresse die Verbindung ausfindig gemacht werden, von der aus das „C“-Paket ursprünglich empfangen wurde. Dadurch kann das O-Paket weitergeleitet werden.

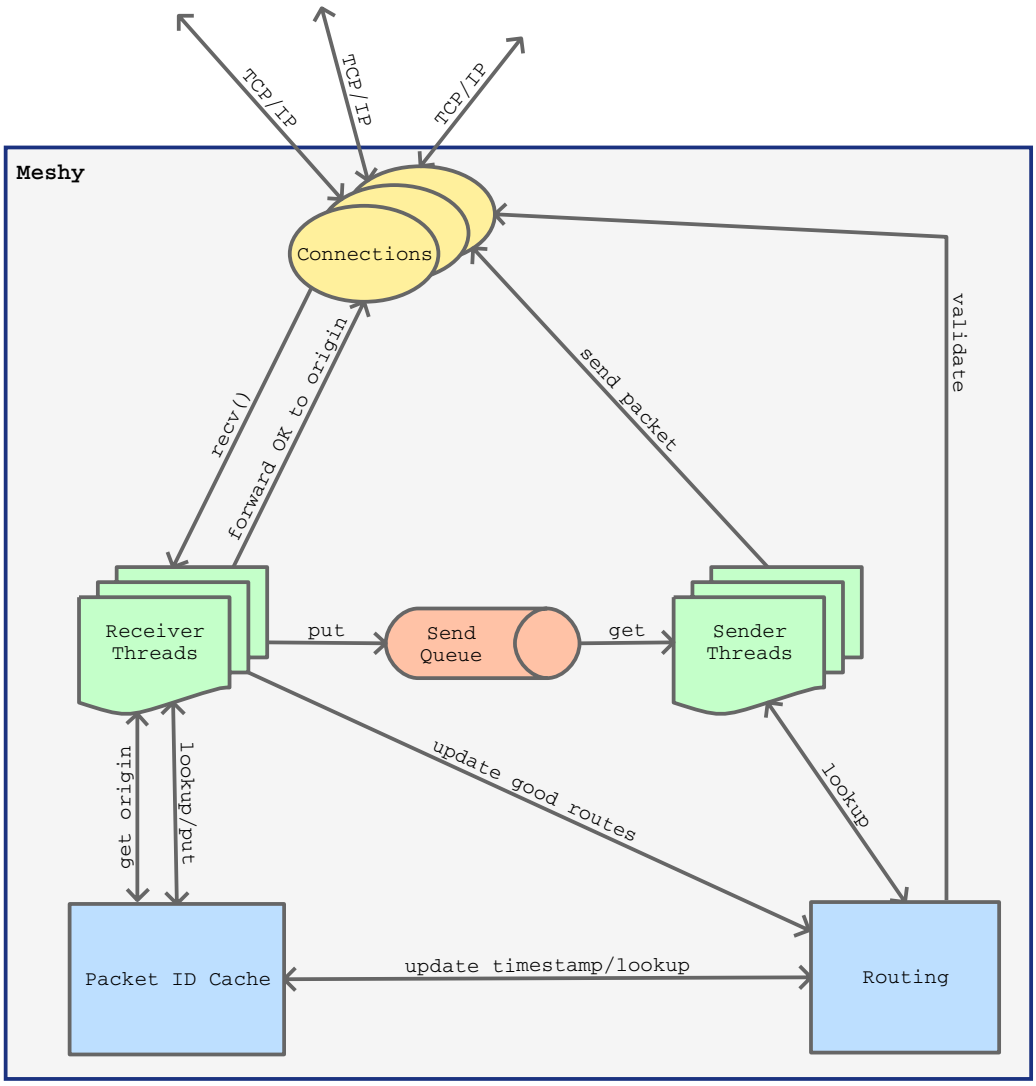

Abbildung 2.1: *Übersicht der Architektur von Meshy*

Es lässt sich erkennen, dass es zwingend notwendig ist, die Paket-ID nie alleine sondern immer in Kombination mit der Adresse (0 oder 1) zu betrachten, da sonst sehr einfach Mesh-Netzwerke konstruiert werden können in denen die Kommunikation nicht in beide Richtungen oder zeitweise gar nicht funktioniert.

2.2.3 Routing

Das „Routing“ ist die Komponente die entscheidet, wohin ein Paket gesendet werden soll. Für jedes C-Paket das gesendet wird, muss das Routing eine gültige Route oder „Broadcast“ als Antwort liefern können. In jedem Fall wird die aktuelle Zeit für die angegebene Adresse hinterlegt. Bei jedem eingehenden O-Paket wird das Routing ebenfalls informiert. Anhand der Zeit kann dann entschieden werden, ob die Route gut ist. Eine „gute“ Route muss immer wieder innerhalb des konfigurierbaren Timeouts durch ein O-Paket validiert werden.

2.2.4 Main Thread

Der `main()`-Thread kümmert sich zuerst um alle nötigen Initialisierungen. Danach wird der Listening-Port aufgesetzt und in einer Endlosschleife auf neue eingehende Verbindungen

gewartet. Für jede neue Verbindung wird ein neuer „Receiver Thread“ erstellt.

2.2.5 Receiver Threads

Die „Receiver Threads“ empfangen die Pakete auf den Verbindungen. Dazu wird ein einer Schlaufe mittels `recv()` blockierend auf Daten gewartet. Die Schlaufe bricht erst ab, wenn `recv()` einen Fehler zurück gibt, sprich wenn die Verbindung getrennt wurde. Die Pakete werden auf den Inhalt überprüft und wie folgt behandelt:

- **C-Paket** (Content)

  Zuerst wird im „ID Cache“ geprüft, ob ein Paket mit derselben ID/Adresse schon einmal empfangen wurde. Ist dies der Fall, wird das Paket verworfen. Im Falle einer neuen ID, wird diese im „ID cache“ gespeichert.

  Ist das Paket im aktuellen Knoten am Ziel angekommen, wird der Inhalt ausgegeben, der Paket-Typ von „C“ auf „O“ geändert und auf der gleichen Verbindung zurückgeschickt.

  Andernfalls wird das Paket zur weiteren Verarbeitung in die „Send Queue“ eingereiht.

- **O-Paket** (OK - Acknowledge)

  Zuerst wird im „ID Cache“ geprüft, ob ein Paket mit derselben ID/Adresse überhaupt als C-Paket empfangen wurde. Ist dies der Fall und ein O-Paket für dieselbe ID/Adresse wurde noch nicht empfangen, gibt der „ID Cache“ die Verbindung zurück auf der das C-Paket ursprünglich empfangen wurde. Das Paket wird ohne Umwege über die Queue direkt an diese Verbindung gesendet.

  Das Routing wird über das O-Paket informiert damit die Route entsprechend angepasst/validiert werden kann.

- **N-Paket** (Neighbor)

  Das Paket enthält die Anweisung eine Verbindung zu einem weiteren Knoten aufzubauen. Die Verbindung wird aufgebaut und durch einen neuen „Receiver Thread“ behandelt.

2.2.6 Send Queue

Die „Send Queue“ ist eine einfache Thread-safe Queue in der Pakete zur weiteren Verarbeitung durch die „Sender Threads“ aufgereiht werden.

2.2.7 Sender Threads

Die „Sender Threads“ sind einfache Worker-Threads die sich zu sendende Pakete aus der Queue entnehmen und anhand des „Routing“ entweder an eine einzelne oder an alle aktiven Verbindungen, ausser die auf welcher das Paket empfangen wurde, sendet.

Kapitel 3

Implementation

Dieses Kapitel beschreibt die Implementation von Meshy. Dabei wird nicht jede Komponente bis ins kleinste Detail durchleuchtet, sondern die interessanten Dinge hervorgehoben. Die beste Dokumentation ist der Code selbst.

> „Talk is cheap. Show me the code.“ – Linus Torvalds

3.1 Buildprozess

Der Buildprozess von Meshy basiert zu einem grossen Teil auf dem Makefile von „Git“. Das Makefile mag für ein so kleines Programm wie Meshy unnötig komplex erscheinen, bietet aber dennoch einige Vorteile:

- Automatische Konfiguration für Linux, OS X (Darwin) und Slowˆ HSolaris
- Minimale Ausgabe. Compileraufrufe werden nicht ausgegeben, ausser wenn die Option „V=1“ angegeben wird. Dies hat den entscheidenden Vorteil, dass Warnungen besser erkennbar sind. Nebst „Git“ hat auch der Linux-Kernel eine ähnliche Ausgabe im Buildprozess.
- Komplett automatische Dependencies auf Header-Dateien. Dazu werden durch den Compiler während dem Compilieren automatisch Make-Rules mit den Dependencies generiert. Die Rules werden jeweils in einem Ordner „.depend“ für jedes Source-File erstellt und automatisch ins Makefile eingebunden.
- Unterstützung von Clang (LLVM), GCC 3.x und höher und LLVM-GCC (Dragonegg)

Der Buildprozess wurde auf Github zur Verfügung gestellt. Fündige Studenten (dem Fork von `https://github.com/tpo/common_network` folgend) konnten ihn somit finden und ebenfalls verwenden.

`https://github.com/dr-itz/common_network`.

3.2 Allgemein

3.2.1 Aufteilung der Dateien

Die Dateien sind nach den Komponenten gegliedert. D.h. es gibt für jede Komponente ein Header mit der API und eine Implementation, siehe Tabelle 3.1. Alle Funktionen in den Headern enthalten eine kurze Dokumentation.

Alle Funktionen der API einer Komponente haben immer den gleichen Prefix im Namen um eine saubere Gruppierung zu erreichen.

Header (.h)	Implementation (.c)	Komponente/Beschreibung
lib/list.h	-	Generische doppelt gelinkte zirkuläre Liste
lib/net.h	lib/net.c	Netzwerk Routinen
lib/utils.h	lib/utils.c	Verschiedene Utility Funktionen
-	meshy.c	main() von Meshy
connection.h	connection.c	Komponente: Connections
idcache.h	idcache.c	Komponente: Packet ID Cache
packet.h	packet.c	Abbildung eines Datenpaketes
receiver.h	receiver.c	Komponente: Receiver Thread
routing.h	routing.c	Komponente: Routing
sender.h	sender.c	Komponente: Sender Thread
sendq.h	sendq.c	Komponente: Send Queue
-	sendmsg.c	Utility Programm zum Senden von Paketen

Tabelle 3.1: *Aufteilung der Dateien in Meshy*

3.2.2 Coding Style

Der Linux Kernel und Git haben einen sehr schönen Coding Style für C. Darum wird dieser verwendet. Siehe: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/CodingStyle

3.2.3 static

Innerhalb einer Implementation wird alles was nicht zur API der Komponente gehört, als `static` deklariert. Dadurch sind diese Funktionen aus anderen Dateien nicht zugänglich. Zudem sind statische Funktionen minimal schneller.

3.2.4 typedef

Alle Strukturen innerhalb einer Implementation die nicht Teil der API ist, werden immer ohne `typedef` definiert. `typedef` wird in der API verwendet (konkret: `connection_t` und `packet_t`). Der Typedef hier bedeutet, dass der Aufrufer nicht direkt auf die Felder der Struktur zugreifen sollen, sondern die API verwenden. `packet.h` z.B. hat eine OO-ähnliche API für den Zugriff auf das Paket.

3.2.5 Generischer Code

Generischer Code befindet sich im Unterverzeichnis „lib“. Dies beinhaltet einige nützliche Funktionen rund um Netzwerk-Kommunikation, einige Utility-Funktionen und eine generische zirkuläre doppelt gelinkte Liste.

Die Liste wurde von ca. 10 Jahren für ein anderes Projekt aus dem Linux-Kernel `list.h` extrahiert und ein wenig abgeändert. Das Spezielle an der Implementation ist, dass es nur ein

Header ist mit einigen Typen, Macros und `static inline` Funktionen. Die `prev` und `next`-Pointer sind Teil einer Struktur `list_head`. Diese Struktur wird jeweils „inline“ in andere Strukturen eingebettet, nämlich immer dort wo eine Struktur ein Teil einer Liste sein soll. Ein wenig ungewöhnlich aber richtig hübsch.

3.2.6 Verwendung von „goto“

Meshy verwendet im Code an einigen Stellen `goto`. Das mag irritierend und falsch wirken, richtig eingesetzt kann aber die Lesbarkeit und die Wartbarkeit von C-Code erhöht und die Code-Duplizierung reduziert und sogar Fehler vermieden werden. Die Verwendung ist ähnlich den vielen `goto`s im Linux-Kernel, die ebenfalls (mit einigen wenigen wirklich hässlichen Ausnahmen) zur Fehlerbehandlung verwendet werden. Siehe „Chapter 7: Centralized exiting of functions“ im Dokument „CodingStyle“ des Linux Kernel.

3.3 Komponenten

Die Liste der hier beschriebenen Komponenten ist unvollständig. Es wird nicht jede Komponente beschrieben.

3.3.1 Connections

Die „Connections“ als Komponente besteht aus einer Mutex geschützten Liste von Verbindungen die dynamisch veränderbar ist. Neue Verbindungen können hinzugefügt, andere gelöscht werden.

Eine Verbindung selbst, definiert als `connection_t` ist eine Struktur mit allem was zu einer einzelnen Verbindung gehört:

- Socket file descriptor
- Remote-Adresse für Debugging-Zwecke. Hier gespeichert, damit nicht für jede Debug-Meldung `getpeername()` aufgerufen werden muss.
- Status der Verbindung: Nicht verbunden, aktiv, geschlossen.
- Pthread Mutex `lock` um den Zugriff auf die Struktur zu schützen.
- Pthread Mutex `send_lock` um den Schreibzugriff auf den Socket zu serialisieren und gleichzeitig Zugriff auf die Struktur zu erlauben.
- Ein Referenzzähler
- Ein `list_head` für die doppelt gelinkte Liste.

Die Verbindungen werden in einer doppelt gelinkten Liste verwaltet. Da eine TCP/IP-Verbindung jederzeit geschlossen werden kann, die Struktur jedoch von mehreren Threads und Kompomenten verwendet werden kann, muss der „Besitz“ jedes „Objektes“ geregelt sein. Dafür ist der Referenzzähler. Jede Funktion einer Komponente die eine `connection_t *` als Parameter erhält und zwischenspeichert (z.B. in einer Liste), muss den Referenzzähler mittels `connection_own()` erhöhen. Der Aufrufer muss in diesem Fall bereits im Besitz einer Referenz sein. Sobald die Verbindung nicht mehr gebraucht und auch nicht mehr referenziert wird,

muss diese mit `connection_release()` wieder freigegeben werden. Nach dem Aufruf dieser Funktion darf nicht mehr auf die Verbindung zugegriffen werden, denn `connection_release()` gibt den Speicher mit `free()` frei sobald der Referenzzähler Null erreicht.

Für einen Broadcast von Paketen müssen alle Verbindungen zum Senden verwendet werden. Da es mehrere Sender Threads gibt die einander nicht blockieren sollen und da während eines Broadcast auch neue Verbindungen eingehen können, kann nicht einfach die ganze Liste mit einem Mutex gesperrt werden um sicher darüber zu iterieren. Um das Problem zu lösen, wird vor einem Broadcast eine Kopie der Liste als Array angelegt in der bereits auf jeder Verbindung ein `connection_own()` ausgeführt wurde. Der Aufrufer kann dann gefahrlos und ohne Mutex über die Liste iterieren. Andere Threads werden dabei nicht blockiert.

3.3.2 Packet ID Cache

Der „Packet ID Cache“ ist als Array in Kombination mit einer statischen Hash-Table ausgelegt. Das Array wird beim Schreiben als Ring verwendet, d.h. Einträge werden nach z.Z. 1024 erhaltenen unterschiedlichen ID/Adressen-Paaren wieder überschrieben. Zur Steigerung der Performance bei einem Lookup wird zusätzlich neben dem Array noch eine einfache Hashtabelle verwendet. Lookups werden an den folgenden Stellen verwendet:

- Einfügen eines neuen ID/Adressen-Paares. Das Einfügen muss zuerst prüfen, ob ein Eintrag bereits existiert um Duplikate abweisen zu können.
- Beim Empfangen eines O-Paketes muss der Ursprung wieder gefunden werden können.
- Routing

Die Hashtabelle ist ein Array von `list_head_t`. Ein Eintrag im Cache enthält ebenfalls einen `list_head_t` damit dieser in den Hash eingefügt werden kann. Wird ein Eintrag durch die Ring-Eigenschaft überschrieben, kann also der Hash-Eintrag ohne einen Lookup gelöscht werden.

3.3.3 Routing

Das „Routing“ muss für jedes C-Paket entscheiden können, wohin es weiter geschickt werden soll. Da es zwei Adressen gibt, 0 oder 1, gibt es zwei Einträge im Routing. Wird ein Paket gesendet und es kommt innerhalb einer konfigurierbaren Zeit kein O-Paket zurück, gilt eine Route als ungültig und es muss für dieselbe Destination wieder ein Broadcast durchgeführt werden. Um dies zu bewerkstelligen, wird nicht etwa ein Timer verwendet sondern mit Timestamps gearbeitet. Die Funktionsweise ist wie folgt:

- **Route anfragen**

  Wird die Route für ein bestimmtes Paket angefragt, werden folgende Schritte ausgeführt:

  1. Im „Packet ID-Cache“ wird für das Paket der Zeitpunkt der Anfrage als Timestamp hinterlegt.
  2. Es wird geprüft, ob Route vorhanden ist und ob diese gültig ist:
     - Es ist eine Verbindung hinterlegt die aktiv ist

   - Die Route wurde zuvor validiert
   - Oder: Die aktive Route ist nicht validiert und die letzte Anfrage ist noch innerhalb des Timeouts
3. Das Validierungs-Flag der Route wird gelöscht
4. Falls die letzte Anfrage ausserhalb des Timeouts liegt, wird der Timestamp der letzten Anfrage auf die Aktuelle Zeit gesetzt

- **Route validieren durch O-Paket**

  Wird ein O-Paket empfangen, wird dies dem Routing zur Validierung der Route mitgeteilt. Dabei werden folgende Schritte ausgeführt:

  1. Zuerst wird anhand des Timestamps im ID Cache geprüft, ob das O-Paket innerhalb des Routing Timeouts angekommen ist. Ist dies nicht der Fall wird keine weitere Aktion ausgeführt.
  2. Wenn die aktuelle Route für die Adresse nicht (mehr) gültig ist, wird die Verbindung auf der das O-Paket angekommen ist, zur neuen Route.
  3. Entspricht die aktuelle Route der Verbindung auf der das O-Paket angekommen ist, wird die Route validiert.

Dieses Vorgehen stellt sicher, dass eine Route immer wieder geprüft wird. Es wird angenommen, dass wenn eine Route zuvor validiert wurde, diese immer noch gültig ist, auch wenn lange Zeit kein Paket mehr für diese Adresse empfangen wurde. Ansonsten würden viel zu viele Broadcast gesendet werden. Die verwendeten Timestamps lösen auf Millisekunden auf.

3.3.4 Send Queue

Die „Send Queue“ ist als einfaches statisches Array implementiert. Mittels Pthread Conditions ist die Queue blockierend für den Leser falls sie leer ist und blockierend für den Schreiber falls sie voll ist.

Kapitel 4

Fazit und Schlusswort

Mit „Meshy“ konnte erfolgreich ein Mesh-Switch in C mit Pthreads implementiert werden, der die Anforderungen an die Seminararbeit erfüllt. Das Programm besteht das vorgegebene Testprogramm und funktioniert mit den Implementierungen von mindestens zwei anderen Studenten zusammen.

Obwohl das Endprodukt nicht ein weiterverwendbares Programm darstellt, hat die Umsetzung dennoch Spass gemacht. Die Implementation selbst ist viel komplexer als es notwendig wäre um die gegebenen Anforderungen zu erfüllen. Die zusätzliche Komplexität hat viel mehr mit Experimentierfreude zu tun. Als gutes Beispiel hierzu dient der „Packet ID Cache“. Der Zusatz der Hashtabelle wäre für die Korrektheit nicht notwendig, bringt aber eine Verbesserung der Performance und war eine gute Übung im Umgang mit den Datenstrukturen.

Das Beste an der Seminararbeit war jedoch, wieder einmal reines C zu programmieren, eine willkommene Abwechslung gegenüber dem alltäglichen Java EE.

Literaturverzeichnis

[1] Buttlar, Dick ; Farrell, Jacqueline ; Nichols, Bradford: *Pthreads Programming: A Posix Standard for Better Multiprocessing*. O'Reilly Media, 1996. – ISBN 978–1–56592–115–3