Humboldt-Universität zu Berlin

Institut für Informatik

Grundlagen der Rechnerkommunikation

Praktikumsbericht

Gruppe „osi“

Matthias Sax und Benjamin Daeumlich

Wintersemester 2007/2008

19. Februar 2008

Inhaltsverzeichnis

1 Einleitung

1.1 Aufbau des Berichtes

In diesem Kapitel werden zunächst einige einleitende Bemerkungen zur Aufgabenstellung und zur Realisierung dieser gegeben. Anschließend wird auf jede Schicht einzeln eingegangen, wobei der Protokollstack von oben (Anwendungsschicht) nach unten (Übertragungsschicht) durchgegangen wird. Es werden jeweils die verwendeten Verfahren und der Aufbau der Protokolldatentypen beschrieben. Letztlich wird noch auf die Netzwerkadapter bzw. die Zugriffsverfahren auf das Übertragungsmedium eingegangen. Im Anhang befinden sich Beschreibungen zu einigen Tests der Verfahren, die Klassenreferenz und die Arbeitsaufteilung.

1.2 Aufgabenstellung

Es war folgende Aufgabenstellung zu bearbeiten:

1. Entwickeln Sie einen Protokollstack mit folgenden Schichten:
   - Zugriff auf das Übertragungsmedium
     - Token-Ring
     - wahlfreier Zugriff angelehnt an CSMA/CD
   - erste Schicht: Übertragung
     - Übertragung der Datenpakete über ein Kabel (Übertragungsmedium)
     - keine Kontrolle der Übertragung auf Fehler oder Überlauf
     - verbindungslos
   - zweite Schicht: Wegewahl
     - suchen eines Weges der Datenpakete vom Absender über unter Umständen mehrere Zwischensysteme (Router) zum Zielsystem
     - verbindungslos
   - dritte Schicht: Transport
     - sichere Übertragung der Datenpakete von einer absendenden Anwendung zu einer empfangenden Anwendung
     - Fehlersteuerung, Flusssteuerung über „Gleitendes Fenster“ (Sliding-Window-Mechanismus)
     - verbindungsorientiert
   - vierte Schicht: Anwendungen
     - Namensdienst
       - Auflösung von Namen in numerische Adressen
     - FileServer und FileClient
       - Zeigen des Inhaltes eines Verzeichnisses vom FileServer
       - Kopieren von Dateien vom FileServer zum FileClient
       - Löschen von Dateien des FileServers
     - Router
       - Austausch von Routing-Informationen zwischen Routern (Routing-Protokoll)
2. Zeigen Sie die Funktionsweise

- der Übertragungssteuerung (Token-Ring und CSMA/CD)
- des Routings (Forwarding)
- des Routingprotokolls an einem Netzwerk mit mindestens zwei Workstations und drei Routern; simulieren Sie Routerausfälle

3. Zeigen Sie die Funktionstüchtigkeit
   - der Transportsteuerung für folgende Fehlerfälle
     - Datenpaket geht verloren
     - Quittung geht verloren
   - der Flusssteuerung der Transportsteuerung
     - simulieren Sie Empfangsengpässe
4. Dokumentieren Sie ihre Arbeit

1.3 Realisierung

Wir haben uns für die Verwendung des vorgegebenen Frameworks entschieden. Dementsprechend ist die Programmiersprache Python.
Netzwerkadressen und MAC-Adressen sind bei uns der Einfachheit halber Integerzahlen.
Zur Vereinfachung wurde weiterhin auf Prüfsummenbildung verzichtet, Bitfehler werden somit nicht behandelt. Ausserdem wird zu jedem Zeitpunkt nur eine Transportverbindung unterhalten. Für das Routing kennt jeder Teilnehmer die Netzwerkadressen seiner Nachbarn bzw. der anderen Rechner in seinem Netzwerk.
Die vertikale Kommunikation zwischen den verschiedenen Schichten erfolgt durch Interface Data Units (IDUs) und die horizontale Kommunikation zwischen Instanzen einzelner Schichten erfolgt durch Protocol Data Units (PDUs). Beide Datentypen enthalten dabei Service Data Units (SDUs), IDUs enthalten Interface Control Information (ICIs) und PDUs enthalten Protocol Control Information (PCIs). Der genaue Aufbau dieser Protokolldatentypen wird später in den entsprechenden Abschnitten erklärt.

2 Anwendungsschicht

Die Anwendungsschicht stellt die oberste Schicht im Protokollstack dar. Es werden verschiedene Dienste bereit gestellt. Der Fileserver und der Fileclient ermöglichen eine Dateiübertragung, der Nameserver löst Namen in Netzwerkadressen auf und der Router sorgt für den Austausch von Routing-Informationen zwischen mehreren Routern.
Die Kommunikation zwischen Routern bzw. zwischen Fileclient und Nameserver erfolgt verbindungslos, da jeweils nur ein bzw. wenige Paket versendet werden und somit der „Overhead“ von Verbingungsaufbau bzw. -abbau zu groß wäre. Zwischen Fileclient und Fileserver wird verbingungsorientiert kommuniziert.

2.1 Fileclient

2.1.1 Beschreibung

Der Fileclient wird mit einem Parameter aufgerufen, welcher den Namen des Fileservers repräsentiert. Dieser wird dann an den Nameserver zur Namensauflösung geschickt („NSREQ“). Die Netzwerkadresse des Nameservers ist dabei in der Konfig-Datei (`konfig.py`) des Fileclients zu finden. Sollte der Fileclient vom Nameserver eine 0 als Netzwerkadresse erhalten, wird der Client beendet, da dann der Servername nicht aufgelöst werden konnte, ansonsten wird die Netzwerkadresse gespeichert und ein Verbindungsaufbau zum Fileserver wird initiiert. Anschließend nimmt der Fileclient Befehle entgegen, bis der Befehl `exit` eingegeben wird. Diese werden an die Netzwerkadresse des Fileservers – die in der Zwischenzeit vom Nameserver zurückgesendet wurde („NSAW“) – geschickt („FSREQ“). Weiterhin nimmt der Fileclient permanent Antworten vom Fileserver entgegen („FSAW“).

2.1.2 Befehle

Es können folgende Befehle eingegeben werden:

`show` Fordert den Inhalt des Verzeichnisses des Fileservers an.

`get filename` Fordert vom Fileserver eine Datei mit Dateinamen `filename` an.

`del filename` Löscht auf dem Fileserver eine Datei mit Dateinamen `filename`.

`exit` Beendet den Fileclient.

Bei anderen Befehlen wird eine Fehlermeldung ausgegeben und eine erneute Eingabe ist möglich.

2.1.3 Format A_PDU()

Der Fileclient sendet Pakete mit der PCI „NSREQ“ bzw. „FSREQ“ und empfängt Pakete mit der PCI „NSAW“ bzw. „FSAW“.

- „NSREQ“-Pakete enthalten als SDU den aufzulösenden Servernamen.
- „NSAW“-Pakete enthalten als SDU die aufgelöste Netzwerkadresse des Servernamen bzw. eine 0, falls der Servername nicht aufgelöst werden konnte.
- „FSREQ“-Pakete enthalten als SDU die Kombination von Kommando und Dateiname (`kommando, dateiname`). Ist das Kommando `show`, ist der Dateiname ein leerer String, ansonsten ist es der zu empfangende bzw. zu löschende Dateiname.

- „FSAW“-Pakete enthalten als SDU die Kombination von Rückgabewert und Dateiname `(result, dateiname)`. Bei einem Fehler (unbekanntes Kommando, zu empfangende bzw. zu löschende Datei exisitert nicht) ist der Dateiname ein leerer String und das Resultat die entsprechende Fehlermeldung. Ansonsten ist das Resultat der Inhalt der Datei bzw. des Verzeichnisses.

2.2 Nameserver

2.2.1 Beschreibung

Der Nameserver wartet nach dem Start auf Anfragen des Fileclients („NSREQ“). Es wird versucht, den so erhaltene Servernamen in eine Netzwerkadresse aufzulösen. Die Servernamen mit zugehörigen Netzwerkadressen werden dabei in einer Datei `dns.tbl` in der Form `servername:netzwerkadresse` gespeichert. Der Nameserver sendet anschließend entweder die aufgelöste Netzwerkadresse oder eine 0 zurück („NSAW“), wobei die 0 bedeutet, dass der Servername nicht aufgelöst werden konnte.

2.2.2 Format A_PDU()

Der Nameserver sendet Pakete mit der PCI „NSAW“ und empfängt Pakete mit der PCI „NSREQ“.

- „NSAW“-Pakete enthalten als SDU die aufgelöste Netzwerkadresse des Servernamen bzw. eine 0 falls der Servername nicht aufgelöst werden konnte.
- „NSREQ“-Pakete enthalten als SDU den aufzulösenden Servernamen.

2.3 Fileserver

2.3.1 Beschreibung

Der Fileserver wartet nach dem Start auf Anfragen des Fileclients („FSREQ“). Diese Anfragen werden entsprechend bearbeitet und die Antworten werden an den Client zurückgeschickt („FSAW“).

2.3.2 Format A_PDU()

Der Fileserver sendet Pakete mit der PCI „FSAW“ und empfängt Pakete mit der PCI „FSREQ“.

- „FSREQ“-Pakete enthalten als SDU die Kombination von Kommando und Dateiname `(kommando, dateiname)`. Ist das Kommando `show`, ist der Dateiname ein leerer String, ansonsten ist es der zu empfangende bzw. zu löschende Dateiname.
- „FSAW“-Pakete enthalten als SDU die Kombination von Rückgabewert und Dateiname `(result, dateiname)`. Bei einem Fehler (unbekanntes Kommando, zu empfangende bzw. löschende Datei exisitert nicht) ist der Dateiname ein leerer String und das Resultat die entsprechende Fehlermeldung. Ansonsten ist das Resultat der Inhalt der Datei bzw. des Verzeichnisses.

2.4 Router

2.4.1 Beschreibung

Der Router stellt ein Protokoll zur Verfügung, welches für den Austausch von Routing-Informationen zwischen mehreren Routern verantwortlich ist. Ziel ist es, Einträge für alle Workstations des gesamten Netzwerkes in der Routingtabelle von jedem Router zu speichern. Zudem soll der Weg möglichst

wenig Kosten verursachen. Da sich unser Protokollstack auf kleinere Netzwerke beschränkt, haben wir uns für diese Variante („Host-Routing“) entschieden.
Das verwendete Routing-Protokoll basiert auf dem Prinzip des Link-State-Routings. Dies funktioniert folgendermaßen (aus „Computernetzwerke“ von Andrew S. Tanenbaum):

1. Die Nachbarn und deren Netzwerkadressen ermitteln
2. Die Übertragungszeit oder die Kosten zu jedem seiner Nachbarn messen
3. Ein Paket zusammenstellen, in dem alles steht, was er gelernt hat
4. Dieses Paket an alle anderen Router senden
5. Den kürzesten Pfad zu allen anderen Routern berechnen

In unserem Fall sind alle Nachbarn und deren Netzadressen und auch die Kosten zu diesen bekannt. Diese Informationen stehen in der Routingtabelle, welche in der Datei `route.tbl` gespeichert ist. Diese Datei enthält Zeilen der Form `adapternummer:default-gateway:zielnetzwerkadresse:kosten`. Die Kosten sind in unseren Beispielen willkürlich festgesetzt worden.
Das Paket („LS“-Paket), welches unter Punkt 3 zusammengestellt werden soll, enthält die eigene Netzwerkadresse, die Netzwerkadressen der Nachbarn, die Kosten des Pfades zu jedem Nachbarn und eine Sequenznummer.
Alle anderen Router werden durch `default-gateway`-Einträge mit Wert 0 identifiziert. Ansonsten ist dieser Wert die Netzwerkadresse eines Routers, über den das Paket an eine Workstation mit einer `zielnetzwerkadresse` geschickt werden soll, oder die eigene Netzwerkadresse, falls die Workstation mit einer `zielnetzwerkadresse` direkter Nachbar dieser Routerinstanz ist. Das zusammengestellte Paket wird nun an alle anderen Router verschickt (dies passiert alle 10 Sekunden).
Es sollten nun „LS“-Pakete von anderen Routern eintreffen. Diese Pakete werden jeweils nach dem Empfang an die anderen Router weitergeleitet (außer an den Sender), damit auch Informationen von nicht direkt benachbarten Routern alle Router erreichen. Damit die Pakete nicht ewig im Netzwerk hin- und hergeschickt werden, werden diese Pakete nur weitergeleitet, falls die Sequenznummer neuer ist, als die in einer Tabelle zwischengespeicherte Sequenznummer (wenn sie nicht neuer ist, war das Paket schon einmal da und wurde auch schon weitergeleitet).
Nach kurzer Zeit sollten nun alle Pakete von allen anderen Routern eingetroffen sein und die Topologie des kompletten Netzwerkes ist bekannt. Durch den Dijkstra-Algorithmus lassen sich nun die kürzesten Pfade zu Zielrechnern (in unserem Fall nur Workstations) berechnen. Die Routingtabelle wird entsprechend aktualisiert.
Um Routerausfällen entsprechend entgegenwirken zu können, wird beim Empfang eines „LS“-Paketes die Empfangszeit zwischengespeichert (bzw. aktualisiert). Sollte nun 60 Sekunden lang kein neues Paket von einem Router eintreffen, wird dessen „LS“-Paket aus dem Speicher gelöscht und die Routingtabelle neu berechnet.

2.4.2 Format A_PDU()

Der Router sendet und empfängt Link-State-Pakete, welche als PCI stets die Zeichenkette „LS“ enthalten.
Die SDU hat die Form `(netzwerkadresse, paketnummer, nachbarliste)`. Die `nachbarliste` enthält Tupel `(nachbarnetzwerkadresse, kosten)`, wobei `kosten` die Kosten für die Benutzung des Pfades `(netzwerkadresse, nachnarnetzwerkadresse)` sind.

2.5 Protokolldatentypen

2.5.1 A_PDU()

Dieser Protokolldatentyp ist für die horizontale Kommunikation zwischen Instanzen der Anwendungsschicht verantwortlich. Das Format der A_PDU() variiert von Anwendung zu Anwendung und ist unter den entsprechenden Beschreibungen zu finden.

2.5.2 T_IDU()

Dieser Protokolldatentyp ist für die Kommunikation zwischen der Anwendungs- und Transportschicht verantwortlich. Die T_IDU() enthält als SDU die A_PDU() und als ICI die Kombination `(netzwerkadresse, type, status)`.
Die Netzwerkadresse `netzwerkadresse` ist entweder die Zielnetzwerkadresse, falls das Paket nach unten gereicht wird, oder die Quellnetzwerkadresse, falls es nach oben gereicht wird.
Der Verbindungstyp `type` ist entweder „UDP“ bei einer verbingungslosen Kommunikation, oder „TCP“ bei einer verbindungsorientierten Kommunikation.
Der Status `status` ist entweder:

- **„NEW“**, falls eine neue Verbindung aufgebaut werden soll
- **„END“**, falls eine Verbindung beendet werden soll
- **„ERROR“**, falls es einen Fehler gab
- **„EST“**, falls Daten hoch- bzw. runtergereicht werden sollen

3 Transportschicht

3.1 Beschreibung

Die Transportschicht ist für die Verwaltung von Transportverbindungen zuständig. Dazu zählen Verbindungsaufbau und -abbau, Flusssteuerung und Fehlersteuerung.
Es gibt außerdem die Möglichkeit, eine verbindungslose Kommunikation durchzuführen. In diesem Fall werden die Pakete lediglich durch die Transportschicht durchgereicht.

3.1.1 Verbindungsaufbau

Der Verbindungsaufbau erfolgt nach einem 3-Wege-Handshake-Verfahren. Zuerst wird ein Verbindungswunsch von einer Instanz der Anwendungsschicht geäußert. Daraufhin wird ein Paket mit dem Flag „SYN“ von Partner1 verschickt. Nach Erhalt dieses Paketes von Partner2 verschickt dieser eine Quittung für den Verbindungsaufbau in einem Paket mit dem Flag „SYN,ACK“. Empfängt Partner1 diese Quittung, ist er verbunden und schickt noch einmal eine Quittung an Partner2. Erhält Partner2 diese Quittung, ist er ebenfalls verbunden. Die Verbindung ist nun erfolgreich aufgebaut und es kann mit der Übertragung von Daten begonnen werden.
Nach dem Senden der „SYN“- bzw. „SYN,ACK“-Pakete wird ein Timer gestartet, welcher gestoppt wird, sobald die Verbindung aufgebaut ist. Ist nach Ablauf des Timers noch keine Verbindung aufgebaut, werden diese Pakete erneut verschickt. Nach 3 Versuchen wird abgebrochen, da man davon ausgehen kann, dass die andere Seite nicht verfügbar ist.

3.1.2 Verbindungsabbau

Der Verbindungsabbau erfolgt analog zum Verbindungsaufbau nach einem 3-Wege-Handshake-Verfahren, die Steuerpakete heißen hierbei allerdings „FIN“- bzw. „FIN,ACK“-Pakete.

3.1.3 Datenübertragung

Bevor eine Datenübertragung beginnt (initiiert von einer Instanz der Anwendungsschicht), wird zuerst überprüft, ob überhaupt eine Verbindung besteht. Ist dies der Fall, werden die Daten von der Anwendungschicht, also die A_PDU(), in mehrere Stücke zerteilt, welche später wieder in A_PDU()s verpackt und versendet werden. Die Anzahl der Stücke wird stets mitgesendet, um das letzte Paket zu identifizieren.
Nun beginnt die eigentliche Datenübertragung, gesteuert durch eine Flusssteuerung bzw. Fehlerkontrolle nach einem Sliding-Window-Mechanismus. Der Sender verwaltet ein Sendefenster, welches Datenpakete mit Flag „DATA“ speichert. Es werden stets nur soviele Datenpakete versendet, wie Platz im Sendefenster ist. Die Datenpakete erhalten weiterhin eine Sequenznummer, welche nach jedem Datenpaket inkrementiert wird. Der Empfänger hingegen verwaltet ein Empfangsfenster, in welchem nun die empfangenen Datenpakete gespeichert werden, sofern sie noch nicht dort erhalten sind (Duplikate werden verworfen). Ist das Empfangsfenster voll, wird ein Quittungspaket mit Flag „ACK“ verschickt, welches die letzte Sequenznummer enthält, bis zu der die Datenpakete schon komplett empfangen wurden. Nach Erhalt dieses Quittungspaketes schiebt der Sender nun sein Sendefenster bis zur entsprechenden Stelle weiter und füllt es wieder auf. Die Anzahl der behandelten Sequenznummer in den entsprechenden Puffern (Sende- bzw. Empfangspuffer) ist doppelt so groß wie die jeweils aktuelle Fenstergröße, welche statisch festgelegt ist. Nach dem Senden der Datenpakete wird ein Timer gestartet, welcher gestoppt wird, sobald ein Datenpaket durch ein Quittungspaket bestätigt wurde. Ist nach Ablauf des Timers das Datenpaket noch nicht bestätigt

worden, wird es erneut verschickt. Nach 5 Versuchen wird abgebrochen, da man davon ausgehen kann, dass die andere Seite nicht mehr verfügbar ist. Gehen nun Datenpakete verloren, werden sie auch nicht quittiert und somit werden sie nach Ablauf des Timers erneut verschickt. Geht ein Quittungspaket verloren, werden die entsprechenden Datenpakete auch erneut verschickt, da sie ebenfalls nicht bestätigt wurden. Der Empfänger erhält dann zwar möglicherweise Datenpakete doppelt, aber diese werden verworfen und die Quittung wird erneut verschickt. Jede Datenübertragung enthält weiterhin eine Transaktionsnummer, sodass Pakete von älteren Transaktionen identifiziert und verworfen werden können.
Sobald alle Datenpakete empfangen wurden, werden die eigentlichen Daten wieder zusammengesetzt und an die Anwendungsschicht übergeben. Die Puffer werden nun wieder geleert, sowohl beim Sender (nach dem Hochreichen der Daten), als auch beim Empfänger (nach dem Senden der Quittung für das letzte Paket).

3.2 Protokolldatentypen

3.2.1 T_PDU()

Dieser Protokolldatentyp ist für die horizontale Kommunikation zwischen Instanzen der Transportschicht verantwortlich. Die T_PDU() enthält als SDU die A_PDU() und als PCI die Kombination `(type, flag, seqnr, wsize, tanr, length)`.
Der Verbindungstyp `type` ist entweder „TCP“ bei einer verbindungsorientierten Kommunikation oder „UDP“ bei einer verbindungslosen Kommunikation. Im zweiten Fall sind alle anderen Paramter leer.
Der Pakettyp `flag` kann folgende Werte annehmen:

- **„SYN“:** erster Teil des 3-Wege-Handshakes zum Verbindungsaufbau
- **„SYN,ACK“:** zweiter Teil des 3-Wege-Handshakes zum Verbindungsaufbau
- **„FIN“:** erster Teil des 3-Wege-Handshakes zum Verbindungsabbau
- **„FIN,ACK“:** zweiter Teil des 3-Wege-Handshakes zum Verbindungsabbau
- **„ACK“:** Quittungspaket
- **„DATA“:** Datenpaket
- **„ERROR“:** Fehlermeldung, wird nach oben gereicht

Die `seqnr` entspricht beim Senden von Paketen der aktuellen Paketnummer, beim Senden von Quittungen ist es die Nummer des quittierten Paketes. Der Parameter `wsize` gibt die aktuelle Fenstergröße an, `tanr` ist die aktuelle Transaktionsnummer und `length` gibt die Anzahl der verschickten bzw. erwarteten Pakete für die aktuelle Transaktion an.

3.2.2 V_IDU()

Dieser Protokolldatentyp ist für die Kommunikation zwischen der Transport- und Vermittlungsschicht verantwortlich. Die V_IDU() enthält als SDU die T_PDU() und als ICI entweder die Zielnetzwerkadresse, falls sie nach unten gereicht wird, oder die Quellnetzwerkadresse, falls sie nach oben gereicht wird.

4 Vermittlungsschicht

4.1 Beschreibung

Die Vermittlungsschicht ist dafür verantwortlich, Daten an den korrekten Adapter zu schicken, falls Daten gesendet oder weitergeleitet werden sollen. Beim Empfang von Daten wird geprüft, ob sie für die aktuelle Instanz bestimmt sind. Falls dies der Fall ist, werden sie an die Transportschicht hochgereicht, falls nicht, werden sie weitergeleitet (Forwarding).
Der korrekte Adapter kann über die Informationen aus einer so genannten Routingtabelle bestimmt werden. Diese ist in einer Datei namens `route.tbl` gespeichert, welche mehrere Zeilen im Format `adapternummer:default-gateway:zielnetzwerkadresse:kosten` enthält. Die Kosten (`kosten`) und das Default-Gateway (`default-gateway`) sind für das Forwarding erstmal uninteressant und werden nur für das Routing-Protokoll benötigt. Über die von der Transportschicht erhaltenen Zielnetzwerkadresse (`zielnetzwerkadresse`) kann nun der Adapter bestimmt werden, an den die Daten geschickt werden sollen. Es exisitert für Workstations stets einen Eintrag mit der Zielnetzwerkadresse 0. An den zugehörigen Adapter werden die Daten geschickt, falls die Zielnetzwerkadresse nicht in den anderen Einträgen der Routingtabelle enthalten ist (d.h. das Ziel ist in einem anderen Netzwerk).

Hier eine kurze Zusammenfassung der Funktionalität der Transportschicht:

- **Senden:**
  - Fall 1 (Daten kommen von Transportschicht): Die Quellnetzwerkadresse ist die eigene Netzwerkadresse und die Zielnetzwerkadresse ist die von der Transportschicht erhaltene Adresse.
  - Fall 2 (Daten werden von der Vermittlungsschicht weitergeleitet): Die Quellnetzwerkadresse ist die ursprüngliche Quellnetzwerkadresse (beim Runterreichen zwischengespeichert) und die Zielnetzwerkadresse ist die ursprüngliche Zielnetzwerkadresse.
  - In beiden Fällen wird der korrekte Adapter bestimmt und die Daten werden an die Übertragungsschicht runtergereicht.
- **Empfangen:**
  - Fall 1 (Daten sind für die aktuelle Instanz bestimmt): Hochreichen der Daten an die Transportschicht
  - Fall 2 (Daten sind nicht für die aktuelle Instanz bestimmt): Quellnetzwerkadresse zwischenspeichern und Daten wieder runterreichen (Forwarding)

4.2 Protokolldatentypen

4.2.1 V_PDU()

Dieser Protokolldatentyp ist für die horizontale Kommunikation zwischen Instanzen der Vermittlungsschicht verantwortlich. Die V_PDU() enthält als SDU die T_PDU() und als PCI die Kombination von Zielnetzwerkadresse und Quellnetzwerkadresse `(zielnetzwerkadresse, quellnetzwerkadresse)`.

4.2.2 U_IDU()

Dieser Protokolldatentyp ist für die Kommunikation zwischen der Vermittlungs- und Übertagungsschicht verantwortlich. Die U_IDU() enthält als SDU die V_PDU() und als ICI die Kombination von Adapternummer und Zielnetzwerkadresse `(adapternr, zielnetzwerkadresse)`. Wenn die U_IDU() nach oben gereicht wird, ist die Zielnetzwerkadresse (`zielnetzwerkadresse`) die 0, da diese Information in dieser Richtung nicht benötigt wird.

5 Übertragungschicht

5.1 Beschreibung

Die Übertragungsschicht ist dafür verantwortlich, Pakete an die richtige MAC-Adresse zu schicken. Beim Empfang von Daten wird weiterhin überprüft, ob sie für die aktuelle Instanz vorgesehen sind. Die MAC-Adressen für Teilnehmer aus dem eigenen Netzwerk und für ein Default-Gateway je Adapter sind bekannt. Diese werden in einer Datei namens `mac.tbl` gespeichert, welche mehrere Zeilen im Format `adapternummer:zielnetzwerkadresse:macadresse` enthält.
Sollen Daten verschickt werden, erhält die Übertragungsschicht von der Vermittungsschicht eine `adapternummer` und `zielnetzwerkadresse`. Dadurch kann die `macadresse` des Ziels bestimmt werden. Ist das Ziel nicht in der Tabelle enthalten (d.h. es befindet sich in einem anderen Netzwerk) gibt es stets einen Eintrag je Adapter für ein Default-Gateway, wobei dafür die `zielnetzwerkadresse` 0 ist. Die zu verschickenden Daten werden nun mit der MAC-Adresse des Zieles versehen und an den Adapter weitergegeben.
Beim Empfang von Daten wird überprüft, ob die Daten an die Vermittlungsschicht hochgereicht werden müssen. Dies ist des Fall, wenn die MAC-Adresse des Zieles, welche im empfangenen Paket vermerkt ist, mit der eigenen MAC-Adresse (zu finden in der Datei `konfig.py`) übereinstimmt. Weiterhin werden die Daten stets hochgereicht, falls es sich beim empfangenden Adapter um einen Punkt-zu-Punkt-Adapter handelt (also um eine direkte Verbindung zweiter Netzwerkteilnehmer). Handelt es sich um einen Token-Ring-Adapter, wird bei hochgereichten Daten zusätzlich eine Empfangsquittung verschickt, ansonsten werden die Daten wieder an den Adapter zurückgegeben. In beiden Fällen geschiet die über eine extra Schnittstelle, so dass die Daten nicht in die Warteschlange der zu sendenden Daten aufgenommen werden muessen. Bei CSMA/CD werden Pakete mit falscher MAC-Adresse einfach verworfen bzw. bei korrekter MAC-Adressen einfach nur hochgereicht (eine Quittung wird nicht erzeugt).

5.2 Protokolldatentypen

5.2.1 U_PDU()

Dieser Protokolldatentyp ist für die horizontale Kommunikation zwischen Instanzen der Übertragungsschicht verantwortlich. Die U_PDU() enthält als SDU die V_PDU() und als PCI die MAC-Adresse des Zieles. Die U_PDUs werden direkt an den Adapter übergeben (ohne in eine IDU verpackt zu werden).

6 Netzwerkadapter

6.1 Token-Ring-Adapter

6.1.1 Erzeugen des ersten Tokens

Beim Einschalten eines Token-Ring-Adapters, weiß der Adapter, dass es vorerst kein Token auf dem Ring gibt. Es wird daher eine Zufallszahl (`double`) zwischen 0 und 1 generiert. Diese Zahl wird dann mittels eines `dummy-token` über den Ring geschickt. Falls der Ring noch nicht geschlossen ist geht das Paket verloren. Es gibt also beim Aktivieren des letzten Adapters im Ring genau ein `dummy-token`.
Empfängt der Adapter ein `dummy-token`, so prüft er die Zahl darin. Ist die Zahl kleiner als seine eigene, so ersetzt er sie durch seine eigene und schickt das Token weiter. Ist die Zahl größer, setzt er seinen Status auf „habe kein Token“ und leitet das `dummy-token` unverändert weiter. Es kann vorkommen, dass der Adapter nun weitere `dummy-token` empfängt – diese werden nun an die Übertragungsschicht hochgereicht (der Adapter unterscheidet nur noch zwischen dem „echten“ Token und Daten). Da jedoch keine gültige MAC-Adresse vorliegt, gibt die Übertragungsschicht die Daten an den Apapter zurück, der diese einfach weiterleitet. Ist die Zahl gleich der eigenen Zahl, so weiß der Adapter, dass alle anderen eine kleinere Zahl gewürfelt haben. Er setzt seinen Status auf „habe Token“ und der Ring ist bereit.
Der Adapter der die größte Zahl gewürfelt hat, bekommet das Token also als erstes. Falls zwei Adapter die gleiche Zahl generieren würden, ist das Verhalten unbestimmt. Dieser Fall ist aber so unwahrscheinlich, dass wir davon ausgeheh können, dass er nie eintreten wird.

6.1.2 Normaler Betrieb

Hat ein Apater das Token, und es liegen Daten zum Senden vor, so werden die Daten verschickt. Der Apapter wartet nun auf den Empfang einer Quittung. Wenn er diese empfangen hat, sendet er das Token an den nächsten Teilnehmer im Ring. Falls der Apater das Token hat, jedoch keine Daten zum Senden vorhanden sind, so wird das Token sofort weitergereicht.
Empfängt der Adapter das Token, so prüft er ob es das leere Token ist, oder nicht. Falls es das leere Token ist, setzt er seinen Status auf „habe Token“ und sendet eventuell Daten. Falls es nicht das leere Token ist und der Adapter auf eine Quittung wartet, so wird geprüft ob die Quittung gültig ist. Ist dies der Fall so kann das Token weitergereicht werden. Wartet der Apapter nicht auf eine Quittung so werden die Daten aus dem Token entnommen und an die Übertragungsschicht hochgereicht. Falls die Daten für diesen Adapter bestimmt waren, so generiert die Übertragungsschicht eine Quittung und reicht sie an den Adapter nach unten. Anderenfalls gibt die Übertragungsschicht die Daten an den Adapter zurück. Der Adapter versendet die Quittung bzw. die zurück gereichten Daten umgehend.

6.1.3 Kommunikationsformate

Der Einfachheit wegen kommunizieren die Adapter ebenfalls mit PDUs. Ein `dummy-token` hat immer eine leere PCI. Die Zufallszahl ist in der SDU hinterlegt. Ein leeres Token enthält als PCI den String `'token'` (die SDU ist leer). Falls Daten versendet werden, wird ein Paket (U_PDU()) aus dem Puffer gelesen und direkt versand (die MAC-Adresse wurde von der Übertragungsschicht in der PCI hinterlegt – die Daten in der SDU).
Eine Quittung (wird von der Übertragungsschicht erzeugt) besteht aus der negativen MAC-Adresse (normale MAC-Adressen sind bei uns immer positiv – so kann die Übertragunsschicht Quittungen

problemlos von Daten unterscheiden). Die SDU einer Quittung enthält eine Kopie der gesendeten Daten.

6.2 Ethernet-Adapter

6.2.1 Normaler Betrieb

Um CSMA/CD zu simulieren wurde der Hub mit einer Logik ausgestattet. Falls ein Adapter einen Sendewunsch hat, so schickt er zuerst ein `busy`-Paket los. Der Hub sendet an alle anderen Teilnehmer ein `wait`-Paket. Der Hub erwartet nun von diesen Teilnehmern die Rücksendung des `wait`-Pakets. Wurde von allen anderen Teilnehmern das `wait`-Paket zurückgesand, so sendet der Hub ein `send`-Paket an die sendewillige Station. Diese sendet nun ihre Daten, die vom Hub verteilt werden. Alle Teilnehmer gehen nach Empfang des Datenpakets in der Grundzustand über.

6.2.2 Kollision - Hub

Falls der Hub mehrere `busy`-Pakete empfängt, so erkennt er eine Kollision. Sobald er von allen Teilnehmern entweder ein `busy`- oder ein `wait`-Paket empfangen hat sendet er an alle ein `col`-Paket. Alle sendewilligen Stationen erkennen die Kollision und legen sich nach dem CSMA/CD Verfahren für eine bestimmte Zeit schlafen. Alle anderen Teilnehmer gehen in den Grundzustand zurück.

6.2.3 Kollision - Adapter (Zusatz)

Nachdem ein `busy`-Paket versendet wurde könnte es sein, dass der Hub im Vorfeld bereits an `wait`-Paket versand hat. Wird diese `wait`-Paket nur empfangen so wird es einfach ignoriert (der Adapter weiß, dass der Hub noch ein `col`-Paket schicken wird).

6.2.4 Empfangen - Adpater (Zusatz)

Empfängt ein Adapter im Grundzustand ein `wait`-Paket so sendet er dieser zurück und wartet auf Daten. Falls er nun ein `col`-Paket empfängt geht er in den Grundzustand zurück. Anderenfalls reicht er das Datenpaket zur Übertragungsschicht hoch und geht dann in den Grundzustand.

6.2.5 Kommunikationsformate

Der Einfachheit wegen kommunizieren die Adapter ebenfalls mit PDUs. Alle Kommunikationspakete (also alle nicht-Daten-Pakete) habe eine leere SDU. Die PCI besteht je nach Paket aus einem der Strings `'busy'`, `'wait'`, `'send'` oder `'col'`. Daten-Pakete (U_PDU()) werden aus dem Puffer gelesen und direkt versand (die MAC-Adresse wurde von der Übertragungsschicht in der PCI hinterlegt – die Daten in der SDU).
Damit der Hub immer weiß an welchem Port ein Teilnehmer hängt, versenden die Workstation beim Start ein `new`-Paket und beim beenden ein `del`-Paket. Die Anzahl der Teilnehmer ist für den Hub eine wichtige Information, damit er weiß auf wie viele Antworten er nach dem Versenden des `wait`-Pakets warten muss.

A Testanwendungen

A.1 Token-Ring-Adapter

Zum Test und zur Veranschaulichung des Token-Ring-Adapters haben wir einen Token-Ring mit 5 Workstations aufgebaut. Diese Test-Anwendung ist im Verzeichnis `ring_client` zu finden.
Die Workstations besitzen die Netzwerkadressen 1 bis 5 und die MAC-Adressen 10 bis 50 (in Zehnerschritten). Die MAC-Adressen aller anderen Teilnehmer sind bekannt für jede Workstation.
Sobald alle Workstations angeschlossen sind, wird vereinbart, wer das Token erhält. Dann sendet die Workstation mit der Netzwerkadresse 3 einen Text an die übernächste Workstation im Ring (Netzwerkadresse 5). Diese Workstation schneidet nach dem Empfang des Textes den letzten Buchstaben ab und sendet den gekürzten Text wiederum an die übernächste Workstation (Netzwerkadresse 2). Dies wird nun wiederholt bis der Text schließlich nur noch eine leere Zeichenkette ist.
Wenn man in der Datei `debug.py` nun die Variable `debug` auf 1 setzt, kann man die richtige Funktionsweise des Token-Ring-Adapters (Verhandlung über Erstbesitz des Tokens, Sendung nur bei Besitz des Tokens, Quittierung des Empfangs, Weiterleitung des Tokens) sehr gut nachvollziehen.

A.2 Ethernet-Adapter

Zum Testen und zur Veranschaulichung des CSMA/CD-Verfahrens haben wir 4 Workstations mittels eines (intelligenten) Hubs verbunden. Diese Test-Anwendung ist im Verzeichnis `ethernet` zu finden.
Die Workstations besitzen die Netzwerkadressen 1 bis 4 und die MAC-Adressen 10 bis 40 (in Zehnerschritten). Die MAC-Adressen aller anderen Teilnehmer sind bekannt für jede Workstation.
Sobald Workstation 1 (bzw. 3) gestartet wird, versucht diese sofort einen String an Workstation 2 (bzw. 4) zu senden. Ist Workstation 2 (bzw. 4) oder der Hub noch nicht aktiv so geht das Paket einfach verloren. Ansonsten empfängt Workstation 2 (bzw. 4) den String, schneidet ein Zeichen ab und schickt den verkürzten String zurück. Workstation 1 (bzw. 3) verhalten sich nun genau so wie Workstation 2 (bzw. 4). Der String wird so lange hin- und hergeschickt bis er leer ist.
Wenn man in der Datei `debug.py` nun die Variable `debug` auf 1 setzt, kann man die richtige Funktionsweise von CSMA/CD (belegen des Medium mittels `busy`, erkennen einer Kollision und „schlafen“) sehr gut nachvollziehen.

A.3 Routing-Protokoll

Zum Test des Routing-Protokolls haben wir ein Netzwerk aus 4 Routern und 3 Workstations aufgebaut. Diese Test-Anwendung ist im Verzeichnis `routing_test` zu finden. Die Konfiguration des Netzwerkes ist auf Abbildung 1 zu sehen.
Nach dem Start aller Teilnehmer werden nun die Routing-Informationen zwischen den 4 Routern ausgetauscht. Pakete, welche vom Client zu einem der Server (o.B.d.A. der Nameserver) geschickt werden, werden nun über Router R3 geroutet, was durch die Einträge `3:5:2:5` in der Routingtabelle von R1 bzw. `3:5:1:5` in der Routingtabelle von R4 deutlich wird. Die Kosten betragen 5. Nach dem „Abschießen“ von R3 wird nach einer Minute die Routingtabelle von R1 und R4 so geändert, dass nun über R2 geroutet wird, da von R3 seit einer Minute kein Link-State-Paket mehr empfangen wurde. Dies wird durch die Einträge `2:4:2:8` in der Routingtabelle von R1 bzw. `4:4:1:8` in der Routingtabelle von R4 deutlich. Man erkennt auch, dass ursprünglich der „günstigere“ Weg ausgewählt wurde, da die Kosten nun 8 betragen (vorher 5).
Wenn man in der Datei `debug.py` nun die Variable `debug` auf 1 setzt, kann man die richtige

Funktionsweise des Routing-Protokolls (Verschicken von Link-State-Paketen, Aktualisierung der Routingtabelle bei Routerausfall) sehr gut nachvollziehen.

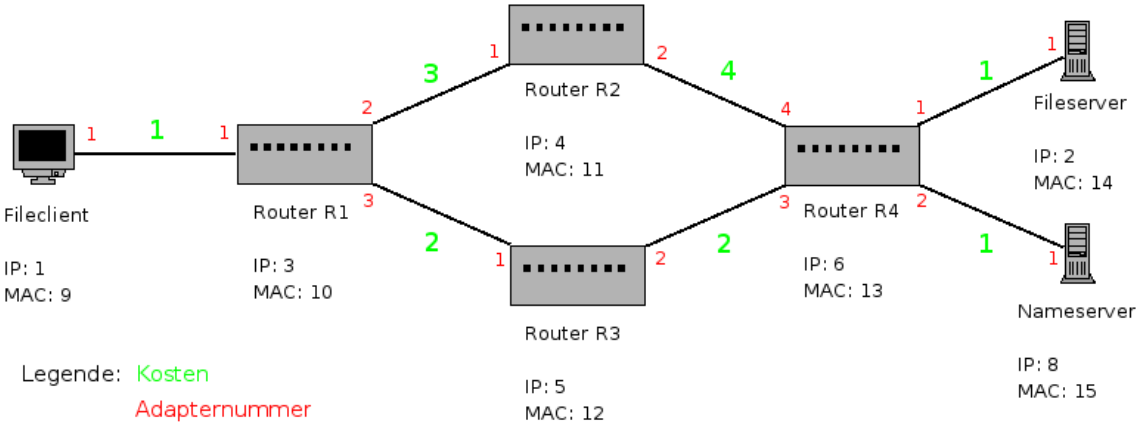

Abbildung 1: Beispielkonfiguration für den Test des Routing-Protokolls

A.4 Transportsteuerung

A.4.1 Fehlerfall

Zum Test der Sendewiederholung im Fehlerfall eignet sich ebenfalls die Testkonfiguration vom Test des Routing-Protokolls. Während einer Dateiübertragung kann man nun R3 „abschießen“. Somit bleiben die Quittungen aus und das Senden des Sendefensters wird alle 15 Sekunden wiederholt. Nach ca. 4 Versuchen haben sich dann schließlich die Routingtabellen entspechend des Routing-Protokolls angepasst und die Sendung wird fortgesetzt.
Für die Simulierung von Fehlern, das heißt dem Verschwinden von Datenpaketen bzw. Quittungen, haben wir eine globale Variable `self._testmode` eingeführt. Ist diese auf „True“ gesetzt, verschwinden Datenpakete bzw. Quittungen entsprechend einer Fehlerrate `self._errorrate`. Es wird vor dem Senden dieser Pakete eine Zufallszahl im Bereich von 0 bis 1 erzeugt. Ist sie kleiner als `self._errorrate`, wird das Paket nicht gesendet. Eine Ausnahme bilden die Pakete zum Verbindungsaufbau bzw. -abbau. Für diese ist die Fehlerrate auf 0.5 festgelegt, damit der Fehlerfall dort möglichst oft Eintritt.
Nach dem Start einer Dateiübertragung in diesem Testmodus kann man sehr schön erkennen, das nicht gesendete Pakete erneut gesendet werden und nicht quittierte Pakete ebenfalls. Es werden außerdem Duplikate erkannt und verworfen.

A.4.2 Flusssteuerung

Bei einer normalen Dateiübertragung kann man die Flusssteuerung sehr gut beobachten.
Es werden stets nur Pakete versendet, falls noch Sendekredit vorhanden ist. Nach dem Empfang von Quittungen wird das Sendefenster stets weitergeschoben. Quittungen werden nur verschickt, wenn das Empfangsfenster voll ist, so wird einer Überlastung des Empfängers entgegengewirkt. Die Größen des Sende- bzw. Empfangsfensters sind statisch festgelegt.

B Klassenreferenz

In diese Abschnitt werden alle von uns veränderten bzw. hinzugefügten Dateien und Klassen mit deren Methoden aufgelistet inklusive einer kurzen Funktionsbeschreibung. Nicht veränderte Dateien, Klassen bzw. Methoden werden dabei nicht erwähnt.

B.1 anwendungsschicht.py

Alle Anwendungen wurden in eine Datei namens `anwendungsschicht.py` gepackt, damit das Einbinden in entsprechende Netzwerkteilnehmer vereinfacht wird und nicht für z.B. zwei verschiedene Router redundanter Code existiert.

B.1.1 Klasse FServer

von_netzwerk(self, t_idu): Extrahiert die A_PDU() aus der übergebenen `t_idu` und prüft ob die PCI dieser A_PDU() „FSREQ“ ist. Wenn dies der Fall ist, wird ein entsprechendes Antwort-Paket („FSAW“) erzeugt und an die Transportschicht übergeben, ansonsten wird das Paket ignoriert.

B.1.2 Klasse NServer

von_netzwerk(self, t_idu): Extrahiert die A_PDU() aus der übergebenen `t_idu` und prüft ob die PCI dieser A_PDU() „NSREQ“ ist. Wenn dies der Fall ist, wird ein entsprechendes Antwort-Paket („NSAW“) erzeugt und an die Transportschicht übergeben, ansonsten wird das Paket ignoriert.

B.1.3 Klasse FClient

von_netzwerk(self, t_idu): Extrahiert die A_PDU() aus der übergebenen `t_idu` und prüft ob die PCI dieser A_PDU() „FSAW“ oder „NSAW“ ist. Ist sie „NSAW“, wird `self.serverip` gesetzt bzw. ein Fehler ausgegeben beim Empfang von 0. Bei einem „FSAW“-Paket wird entweder die empfangene Datei geschrieben oder eine Ausschrift (Fehler, Mitteilung oder Verzeichnisauflistung) ausgegeben. Ansonsten wird das Paket ignoriert.

start_anwendung(self): Zuerst wird ein „NSREQ“-Paket erzeugt, welches für den Nameserver bestimmt ist. Dieses enthält den aufzulösenden Servernamen. Danach wird bis zur Eingabe des Befehls `exit` auf die Eingabe von Befehlen gewartet. Ist der Befehl gültig, wird ein „FSREQ“-Paket erzeugt, welches für den Fileserver bestimmt ist. Ansonsten erscheint eine Fehlerausschrift.

B.1.4 Klasse Router

__init__(self, pstack): Hier werden zuerst verschiedene Tabellen und Variablen initialisert. Danach wird die Routingtabelle aus der Datei `route.tbl` eingelesen. Alle Elemente, bei denen der zweite Eintrag eine 0 ist (Default-Gateway), werden in ein Dictionary der Form `default_gateway:interface` eingetragen. Der Parameter `pstack` wird benötigt, um später die Routingtabelle der Vermittungsschicht überschreiben zu können.

von_netzwerk(self, t_idu): Extrahiert die A_PDU() aus der übergebenen `t_idu` und prüft ob die PCI dieser A_PDU() „LS“ ist. Trägt in diesem Fall neue Link-State-Pakete in das

Dictionary der Link-State-Pakete `self._linkstatepackages` ein bzw. aktualisert die Zeit bei vorhandenen Link-State-Paketen (aktualisiert gegebenenfalls die Routingtabelle mittels `_update_table(self)`). Ansonsten wird das Paket ignoriert.

start_anwendung(self): Sendet alle 10 Sekunden ein „LS“-Paket an alle Nachbarrouter (mittels `_an_default_gateways(self, a_pdu, source)`). Zudem wird das eigene Paket in das Dictionary der Link-State-Pakete `self._linkstatepackages` eingetragen. Es wird ebenfalls überprüft, ob für ein anderes Link-State-Paket die Zeit abgelaufen ist. Ist dies der Fall wird es auf dem Dictionary entfernt und die Routingtabelle wird aktualisiert (mittels `_update_table(self)`).

_an_default_gateways(self, a_pdu, source): Sendet ein „LS“-Paket `a_pdu` an alle Nachbarrouter außer an die Quelle `source`.

_update_table(self): Aktualisert die Routingtabelle und überschreibt am Ende die Tabelle der Vermittlungsschicht mit der aktualisierten Tabelle. Extrahiert dazu die Knoten und Kanten aus dem Dictionary der Link-State-Pakete `self._linkstatepackages` und führt anschließend den Dijkstra-Algorithmus (`dijkstra.py`) aus. Es werden nur Routingeinträge für Workstations in die Routingtabelle eingetragen.

B.2 transportschicht.py

Im Folgenden ist mit „Senden“ stets die Übergabe von Paketen an die Vermittlungsschicht gemeint.

__init__(self, kanal): Initialisiert Puffer und weitere Variablen.

sendTimeOut(self, seqnr, v_idu): Wird beim Senden eines Datenpaketes `v_idu` mit der Sequenznummer `seqnr` aufgerufen (in einem Thread). Nach dem ersten Senden wird ein Timer von 15 Sekunden gestartet. Sollte das gesendete Paket bis zum Ablauf des Timers nicht quittiert worden sein, wird es erneut gesendet. Nach 5 Versuchen wird die Sendung abgebrochen.

connectionTimeOut(self, v_idu): Hat die gleiche Funktion wie `sendTimeOut()`, nur dass die Methode beim Versenden der Pakete bei Verbindungsaufbau aufgerufen wird (in einem Thread). Sollte bis zum Ablauf des Timers die Verbindung nicht aufgebaut sein, wird das entsprechende Paket `v_idu` erneut gesendet. Nach 3 Versuchen wird abgebrochen.

closeTimeOut(self, v_idu): Hat die gleiche Funktion wie `sendTimeOut()`, nur dass die Methode beim Versenden der Pakete bei Verbindungsabbau aufgerufen wird (in einem Thread). Sollte bis zum Ablauf des Timers die Verbindung nicht aufgebaut sein, wird das entsprechende Paket `v_idu` erneut gesendet. Nach 3 Versuchen wird abgebrochen.

von_n_minus_1(self, v_idu): Extrahiert die Quellnetzwerkadresse aus der `v_idu` und die Parameter für die Transportschicht aus der `t_pdu` (in der `v_idu` enthalten). Bei einer verbindungslosen Kommunikation werden die Daten in eine T_IDU() eingepackt und eine Schicht nach oben gereicht. Bei einer verbindungsorientierten Kommunikation werden die Parameter ausgewertet und es wird entprechend auf diese reagiert (Pakete für Verbindungsaufbau bzw. -abbau senden, Daten senden, Quittungen senden).

von_n_plus_1(self, t_idu): Extrahiert die Zielnetzwerkadresse, die Verbindungsart und den Verbindungsstatus aus der `t_idu`. Bei einer verbindungslosen Kommunikation wird lediglich die Zielnetzwerkadresse die ICI einer V_IDU() eingetragen, welche dann eine Schicht nach unten

gereicht wird. Bei einer verbindungsorientierten Kommunikation wird entweder ein Verbindungsaufbau, ein Verbindungsabbau oder eine Datenübertragung initiiert.

senddata(self, wsize, seqnr, dest): Sendet Datenpakete aus dem Sendepuffer (ausgehend von der Sequenznummer `seqnr`) an `dest`, falls noch Sendekredit vorhanden ist im Fenster der Größe `wsize`.

sendack(self, wsize, seqnr, dest): Sendet Quittungen für Pakete (ausgehend von der Sequenznummer `seqnr`), falls bereits genug Datenpakete (`wsize`) zum Quittieren gesammelt wurden, an `dest`.

teile(self, a_pdu, size): Wandelt `a_pdu` in einen Objektstring (mittels `pickle.dumps()`) um und teilt diesen in Teile der Größe `size` auf. Diese Teile werden in einem Array verpackt zurückgegeben.

fuege(self, adata): Erzeugt aus dem Array `adata` einen String, wandelt diesen in eine A_PDU() um (mittels `pickle.loads()`) und gibt diese zurück.

B.3 vermittlungsschicht.py

__init__(self, kanal, nw_adresse=None): Setzt die eigene Netzwerkadresse `self.netzwerkadresse` auf den Parameter `nw_adresse` und liest die Routingtabelle aus der Datei `route.tbl` ein und speichert die Einträge in einem Array `self._table`.

von_n_minus_1(self, u_idu): Extrahiert Ziel- und Quellnetzwerkadresse aus der `u_idu` und prüft, ob die Daten für die aktuelle Instanz bestimmt sind. Ist dies der Fall, werden sie eine Schicht hochgereicht (in Form einer V_IDU(), welche als ICI die Quellnetzwerkadresse enthält), wenn nicht werden sie wieder runtergereicht (in diesem Fall wird die ursprüngliche Quellnetzwerkadresse zwischengespeichert).

von_n_plus_1(self, v_idu): Bei wieder runtergereichten Daten muss zuerst die ursprüngliche Quellnetzwerkadresse wiederhergestellt werden, ansonsten entspricht sie der eigenen Netzwerkadresse. Anschliessend wird aus der Routingtabelle der Adapter bestimmt, an den die Daten gesendet werden müssen. Danach wird eine U_IDU(), welche als ICI die Adapternummer enthält, eine Schicht nach unten gegeben.

B.4 uebertragungsschicht.py

__init__(self): Liest die MAC-Tabelle aus der Datei `mac.tbl` ein und erzeugt zwei Dictionaries. Das Dictionary `self._standard_mac` hat die Form `interface:mac` und ist für die MAC-Adressen der Default-Gateways verantwortlich, das Dictionary `self._mac` hat die Form `(interface:ziel):mac` und beinhaltet die MAC-Adressen für bekannte Ziele.

von_adaptern(self, kanal): Holt sich eine U_PDU() vom Kanal und extrahiert die MAC-Adresse des Ziels aus dessen PCI. Entscheidet anschließend, ob die Daten eine Schicht nach oben gereicht werden (in einer U_IDU(), wobei die ICI die Adapternummer des empfangenen Adapters ist). Dies ist dann der Fall, wenn es sich beim Adapter um einen Punkt-zu-Punkt-Adapter handelt oder wenn die extrahierte MAC-Adresse mit der eigenen MAC-Adresse übereinstimmt. Bei „falschen“ Daten und bei darunterliegendem Token-Ring-Adapter werden die Daten zur Weiterleitung an den Apapter zurück gegeben. Bei korrekten Daten und

Verwendung eines Token-Ring-Adapters, wird eine Quittung erzeugt. Bei CSMA/CD werden Pakete mit falscher MAC-Adresse einfach verworfen bzw. bei korrekter MAC-Adressen einfach nur hochgereicht.

von_n_plus_1(self, v_idu): Bestimmt die MAC-Adresse des Ziels und trägt diese in die PCI der U_PDU() ein und gibt diese dann an den Adapter weiter.

B.5 token_ring_adapter.py

__init__(self): Setzt den Adaptertyp auf `token_ring` (zur Abfrage für die Übertragungsschicht) und den Tokenstatus auf „noch nicht vorhanden“.

_verbinde(self): Testet ob der Adapter an 2 Kabel angeschlossen wurde.

_wait_for_first_token(self): Wird von `_an_kabel()` gerufen. Geht in eine Schleife, bis der eigene Token-Status gesetzt ist.

_gen_first_token(self): Wird von `_von_kabel()` gerufen. Erzeugt ein `dummy-token` das auf den Ringe gelegt wird. Empfängt `dummy-token` bis Anfangsstatus gesetzt werden kann (entweder „habe Token“ oder „habe kein Token“).

_von_kabel(self, nr, kabel): Ruft beim Start einmalig `_gen_first_token()`. Liest Pakete vom Kabel und gibt Daten zusammen mit der Adapternummer an die Übertragungsschicht weiter. Verändert den Adapterstatus beim Empfang des Leertokens oder einer für diesen Adapter bestimmten Quittung.

_an_kabel(self, kabel): Ruft beim Start einmalig `_wait_for_first_token()`. Versendet genau ein Datenpakete bei vorhandenem Token. Reicht Token nach Empfang einer Quittung oder keinen Daten zum Senden weiter. Leitet Daten und Quittungen die nicht für diesen Adapter bestimmt sind weiter.

B.6 ethernet_adapter.py

__init__(self): Setzt den Adaptertyp auf `ethernet` (zur Abfrage für die Übertragungsschicht).

_verbinde(self): Setzt den Adpater in den Grundzustand.

_von_kabel(self, nr, kabel): Empfäng Pakete vom Kabel und überprüft ob es sich um Steuerpakete oder Daten handelt. Setzt bei empfangenen Steuerpaketen (`'wait'`, `'send'`, `'col'`) den Adapter in den entsprechenden Zustand. Datenpakete werden zusammen mit der Adapternummer zur Übertragungsschicht hochgereicht.

_an_kabel(self, kabel): Meldet sich beim Start einmalig beim Hub an. Prüft dann immer ob Daten zum Senden vorhanden sind und versucht diese zu Senden. Verschickt (falls es der aktuelle Zustand erlaubt) `'busy'`- oder Datenpakete. Legt sich bei Kollisionen entsprechend CSMA/CD „schlafen“. Meldet sich beim Beenden wieder beim Hub ab.

B.7 ehub.py

Die Klasse `ehub.py` entspricht im Wesentlichen der vorgegebenen Klasse `hub.py`, nur dass Mechanismen für die Verwendung von Ethernetadaptern hinzugefügt wurden, wie z.B. die Erkennung von Kollisionen.

B.7.1 Klasse EPort

__init__(self): Weist dem Port eine ID zu.

_lies_kabel(self): Liest Daten vom Kabel und speicher sie zusammen mit der Port-ID in der Inputqueue.

_schreib_kabel(self), connect(self, kabel) und disconnect(self) sind unverändert (im Vergleich zu hub.py).

B.7.2 Klasse EHUB

__init__(self): Initialisiert pro Port eine Input- und eine Outputqueue.

start(self) und stop(self) sind unverändert (im Vergleich zu hub.py).

_doit(self): Verwaltet die Anzahl der aktuellen Teilnehmen im LAN. Teilnehmer können sich mittels `'new'`-Pakten an- und `'del'`-Pakten abmelden. Empfängt Steuerpakete der Teilnehmer und erkennt Kollisionen. Verschickt Steuerpakete an Teilnehmer und verteilt Datenpaket.

B.8 dijkstra.py

Die Klasse Dijkstra ist für die Berechnung des kürzesten Pfades in einem Graphen verantwortlich. Dieser wird für das Routing-Protokoll benötigt. Der verwendete Pseudocode ist im Quelltext zu finden.

__init__(self, knoten, kanten): Setzt die Knoten und Kanten (einmal ohne und einmal mit Gewicht). Der Paramter `knoten` ist ein Array von den Knoten des Graphen und der Parameter `kanten` ist ein Dictionary der Form `(u,v):kosten`. Dies bedeutetet, dass der Pfad von Knoten `u` nach Knoten `v` die Kosten `kosten` hat.

shortest_path(self, start, ziel): Berechnet den kürzesten Pfad von einem Knoten `start` zu einem Knoten `ziel`. Zurückgegeben wird die Kombination `(pfad, kosten)`, wobei `pfad` ein Array der Form `[start, z1, ... , zn, ziel]` ist und `kosten` die Kosten des kürzesten Pfades von `start` nach `ziel` repräsentiert.

B.9 debug.py

Diese Klasse gibt bei auf 1 gesetzer Variable `debug` Debuginformationen aus (beim Aufruf der Methode `dprint(s1, s2)`). Setzt man diese Variable auf 0, wird nichts ausgegeben.

dprint(s1, s2): Gibt bei gesetzter Variable `debug` eine Fehlermeldung aus (inklusive Zeit), wobei `s1` die Schicht und `s2` die eigentliche Fehlerausschrift angeben soll.

C Arbeitsnachweis

C.1 Benjamin Daeumlich

- Anwendungsschicht
- Transportschicht
- Vermittlungsschicht
- Übertragungsschicht

C.2 Matthias Sax

- Vermittlungsschicht
- Übertragungsschicht
- Token-Ring-Adapter
- Ethernet-Adapter