Eidgenössische
Technische Hochschule
Zürich

Topsy TCP/IP

Implementation des TCP-Layers und des Socket-Interfaces

Semesterarbeit von

Reto Gähler, Toni Kaufmann

Datum:	4. Februar 2000
Dauer:	Wintersemester 1999/2000
Supervisor:	David Schweikert, George Fankhauser
Professor:	Bernhard Plattner
Dateiname:	Report.doc

Abstract

Topsy (Teachable operating system) is a portable micro-kernel operating system designed for teaching purposes at the ETH Zurich. Goal of this project was the design and implementation of a TCP/IP stack for Topsy. The networking infrastructure of Topsy provided already UDP, IP, ICMP and ARP. TCP and a unix-like socket-interface had to be implemented.

The XINU TCP-implementation from Douglas E. Comer and David L. Stevens [DC2] was taken as base. It supports the whole TCP-standart in a very compact and clear way.

The implementation works with an input-, an output- and a timer-process. The timer-process was embedded in the existing operating system as a general purpose timer. The whole source-code was written completely new with good readable notation rules for variable names. The code was adapted to fit in the existing network structure as well as to the Topsy specific memory- and thread-management. For testing and presentation purposes an echo-deamon and a tiny web-server application were implemented upon the socket- interface.

Zusammenfassung

Topsy (Teachable operating system) ist ein portables Microkernel-Betriebssystem, das an der ETH Zürich für den Unterricht entworfen wurde. Ziel dieser Semesterarbeit war der Design und die Implementation eines TCP/IP Stacks für Topsy. Topsy verfügte bereits über die Protokolle UDP, IP, ICMP und ARP. TCP und ein Unix-ähnliches Socket-Interface mussten implementiert werden.

Die XINU TCP-Implementation von Douglas E. Comer und David L. Stevens [DC2] wurde als Basis genommen. Sie unterstützt den gesamten TCP-Standard in einer sehr kompakten und einfachen Form.

Die Implementation arbeitet mit je einem Input-, Output- und Timer-Prozess. Der Timer-Prozess wurde als General Purpose Timer in Topsy integriert. Der gesamte Source-Code wurde komplett neu geschrieben mit gut lesbarer Notation für Variablennamen. Der Code wurde einerseits an die existierende Netzwerkstruktur angepasst wie auch an das Topsy-spezifische Speicher- und Thread-Management. Für Test- und Präsentationszwecke wurden eine Echo-Deamon sowie eine kleine Webserver Applikation über dem Socket-Interface implementiert.

Zürich, 3. Februar 2000

Reto Gähler Toni Kaufmann

INHALTSVERZEICHNIS

1 Einführung

1.1 Notation

In diesem Dokument werden folgende Notationen und Schriftgrade verwendet:

[AAA]	Literaturhinweis
`Code`	`Code Beispiele sind in Courier 9 dargestellt`
Funktionen	*Funktionen und Namen von Variabeln und Datentypen sind in italic dargestellt*

Tabelle 1.1-1, Notation

1.2 Dokumentbeschreibung

Das Dokument beschreibt im Wesentlichen nur die Implementation von TCP und des Socket-Layers in Topsy. Weitergehende Information zu Topsy und den Netzwerkprotokollen können aus [GFR] und [DWS] entnommen werden.

Kapitel 1:	Gibt eine Einführung in das Projekt und beschreibt den Ist-Zustand der TCP/IP Implementation in Topsy
Kapitel 2:	Entscheidungsgrundlagen für die Art der TCP-Implementation
Kapitel 3:	Die eigentliche Implementation und die Beschreibung deren Module
Kapitel 4:	Beschreibung des Socket-Layers und wie er angewendet wird anhand von Beispielen
Kapitel 5:	Inbetriebnahme und Tests der TCP-Implementation
Kapitel 6:	Schlussfolgerungen, Erfahrungen sowie Verbesserungsvorschläge
Kapitel 7:	Im Anhang findet sich das Literaturverzeichnis sowie die nicht ins Dokument eingebundene Aufgabenstellung

1.3 Projektübersicht

Topsy ist ein portables Microkernel Betriebssystem, das am TIK für den Unterricht entworfen wurde. In der ersten Version wurde es für die Familie der 32-bit MIPS Prozessoren gebaut. Es zeichnet sich durch eine saubere Struktur, eine hohe Portabilität (Trennung des Systems in hardware-abhängige und -unabhängige Module) und eine gute Dokumentation aus. Im weiteren wird das System auf die Familie der intel i386 Prozessoren portiert. Weitere Dokumentation über Topsy ist unter http://www.tik.ee.ethz.ch/~topsy verfügbar.

David Schweikert implementierte in einer Semesterarbeit bereits die Protokolle IP, ICMP, ARP sowie UDP [DWS].

1.4 Entwicklungsumgebung

Für die Entwicklung kann Topsy auf einem JavaSimulator, der den MIPS-Prozessor emuliert, gestartet werden. Diese Java-Simulation wie auch der MIPS-Compiler laufen auf einer Linux-Station. Die TCP/IP-Kommunikation von Linux zu Topsy läuft über ein Ethertap-Device auf Linux.

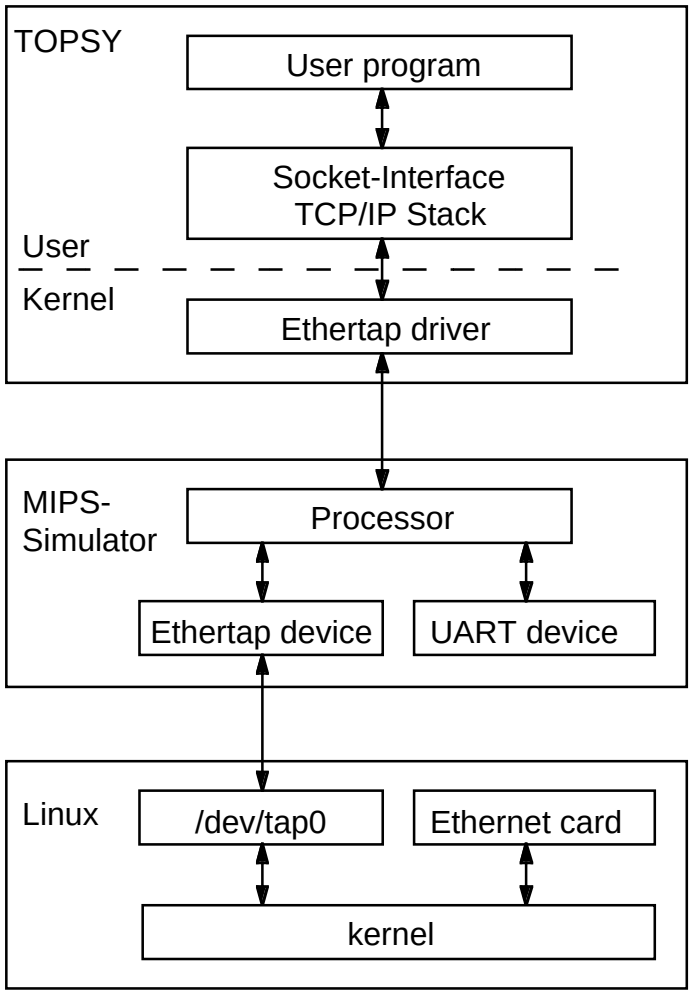

Abbildung 1.4-1, Entwicklungsumgebung

1.4.1 Debug-Möglichkeiten

Es steht ein Modul NetDebug zur Verfügung, über das Textstrings auf die Konsole ausgegeben werden können.

1.5 Entwicklungsstand Topsy Net Version 1.1

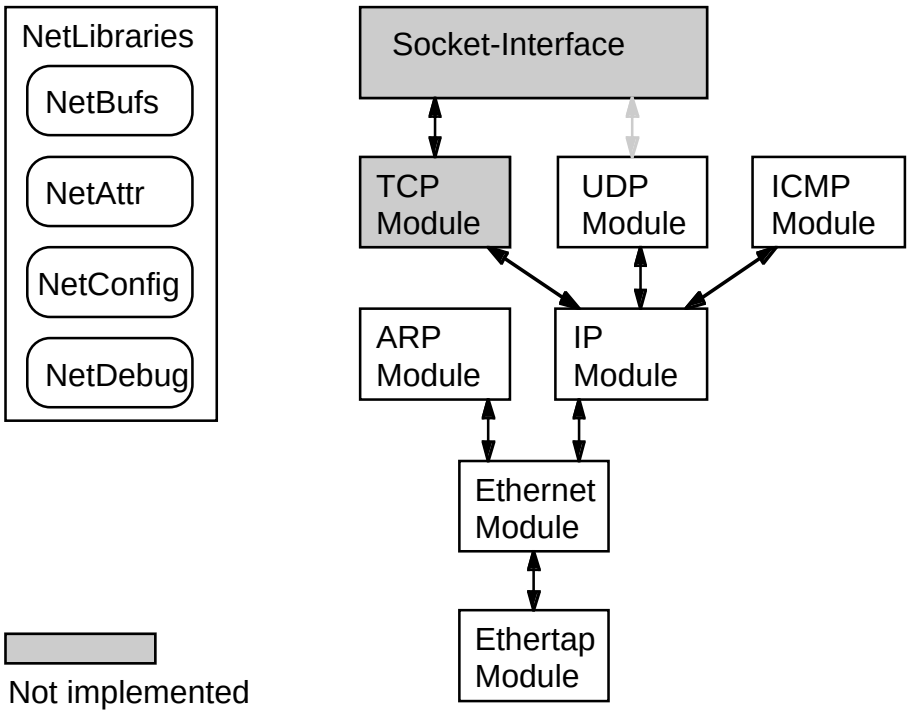

Abbildung 1.5-1, Topsy Net

Alle weissen Net-Module stehen in Topsy schon zur Verfügung. Die grau hinterlegten Module TCP und Socket sind Aufgabe dieser Semesterarbeit.

2 Software-Analyse

2.1 TCP-Implementation

Es stellte sich die Frage, ob TCP vollständig neu programmiert oder eine bestehende Implementation portiert werden sollte. Eine Neuimplementation könnte ideal in Topsy und dessen Struktur integriert werden. Bestehende Implementationen sind mit der Zeit gewachsen und verfügen teilweise über einen beträchtlichen Overhead. Dafür sind sie getestet und bewährten sich im Praxiseinsatz.

2.1.1 Neuimplementation

Die vollständige Implementierung des TCP-Standards ist sehr aufwändig und muss intensiv getestet werden. In bestehenden Implementationen sind auch Praxiserfahrungen eingeflossen, die nicht einfach aus dem Standard entnommen werden können.

2.1.2 Portierung

Es wurden zwei bestehende und gut dokumentierte Implementationen angeschaut.

2.1.2.1 BSD-Unix TCP

TCP in BSD-Unix ist sehr umfangreich und seit Bestehen kontinuierlich angewachsen. Für Topsy dürfte diese Implementation jedoch einen Overkill darstellen.

2.1.2.2 XINU TCP

XINU ist ein schlankes Schulbetriebssystem basierend auf Unix. Die TCP-Implementation ist bedeutend schlanker als die von BSD (rund drei mal weniger Code). Die Implementation basiert auf je einem Sende- und Empfangs- sowie einem Timer-Prozess. Der Code wird in verschiedenen kommerziellen Anwendungen eingesetzt. Die einzige Einschränkung für die Verwendung des Codes ist, dass er nicht in einem Buch publiziert werden darf.

2.2 Design-Entscheid

Der Entscheid fiel auf die Verwendung von XINU TCP als Basis für die Implementation. Bei der Portierung wurde versucht, TCP möglichst gut in Topsy zu integrieren. So wurde jede Zeile Code neu geschrieben und durchwegs die Topsy-NetBufs bzw. –NetAttrs verwendet. Es wurden konsequent die im Kapitel Namenskonventionen beschriebenen Regeln eingehalten.

2.3 Timer

TCP benötigt einen effizienten Timer, der in Topsy nicht vorhanden ist. Der Timer wurde nicht einfach für TCP portiert, sondern in Topsy als allgemein zur Verfügung stehender Timer integriert.

2.4 Plattform für Portierung

Aus verschiedenen Gründen wird die Portierung auf einem Macintosh mit Metrowerks Compiler durchgeführt. Einerseits ist die Entwicklungsumgebung bekannt und andererseits verfügt sie über einen komfortablen Source-Level Debugger. Dazu mussten alle Module, auf die TCP zugreift, von Topsy auf MacOS portiert werden.

3 Software-Implementation

3.1 Modulübersicht (Sourcefiles)

Das folgende Bild zeigt die neuen Module von Topsy grau eingefärbt. Der in der jeweiligen Box angegebene Namen entspricht dem Name der Source Datei mit der zusätzlichen Endung «.h» bzw. «.c»

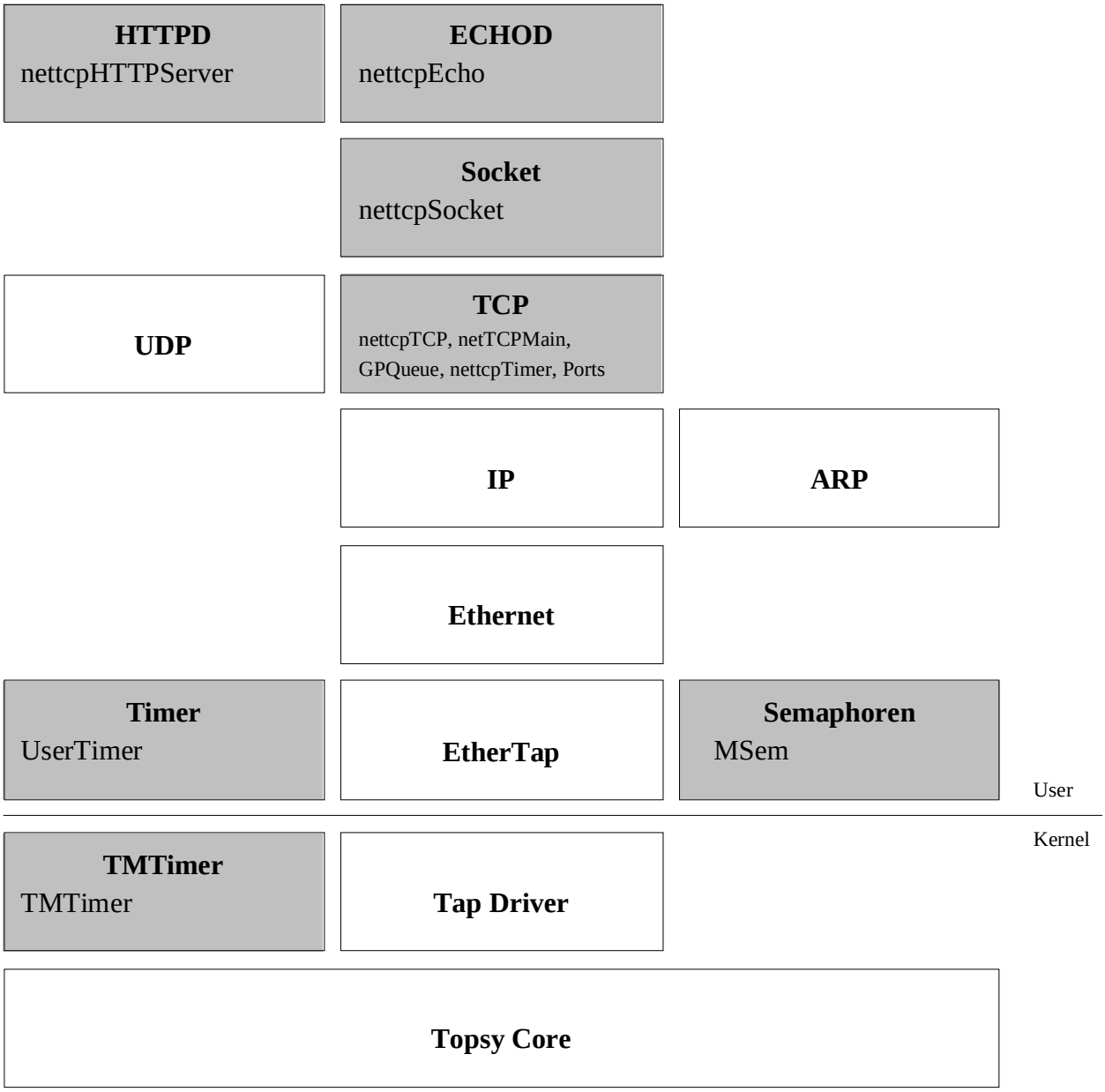

Abbildung 3.1-1, Modulübersicht

3.2 Namenskonventionen

Um die Lesbarkeit des Codes zu erhöhen, wurden Namenskonventionen verwendet. Obwohl diese Technik eine weite Verbreitung hat [SMC], gibt es keinen Standard. Daher sind nachfolgend die für das Topsy TCP Projekt geltenden Konventionen aufgelistet.

Typ	**Abbkürzung**	**Beispiel**
unsigned char	u8	u8Code
char	n8	n8Key
unsigned	u	uLength
int	n	nLength
unsigned short	u16	u16Length
short	n16	n16Error
unsigned long	u32	u32Length
long	n32	n32Size
Boolean	b	bDone
struct	rec	recSocket
enum	set	setConnection

Tabelle 3.2-1, Namenskonvention

Der Speichertyp einer Variable wird vor die eigentliche Typbezeichnung geschrieben

Speichertyp	**Abbkürzung**	**Beispiel**
Pointer	p	pu8Buffer
globale Variable	g	gu32NumOfErrors

Tabelle 3.2-2, Speichertyp

3.3 TCP

Die TCP Implementation in Topsy wurde so weit als möglich von XINU übernommen. In diesem Kapitel wird kurz erklärt wie die Implementation funktioniert und was bei der Portierung von XINU nach Topsy geändert werden musste.
Die eigentliche TCP Protokoll Implementation konnte vollständig übernommen werden und ist in [DC2] ausführlich und komplett beschrieben. Deswegen wird in diesem Dokument nur teilweise darauf eingegangen.

3.3.1 Thread-Modell

Die TCP Implementation verwendet für ihre Dienste drei Threads: Ein Input-Thread, welcher alle eingehenden Daten verarbeitet, ein Output-Thread, welcher für das Senden von Daten verantwortlich ist und ein Timer-Thread. Der Input- und Output-Thread werden in der Funktion *nettcpInit* im *nettcpMain* Modul gestartet. Der Timer-Thread wird im gleichen Zug wie die Shell aufgesetzt.

Das folgende Bild zeigt das Zusammenspiel der drei Prozesse:

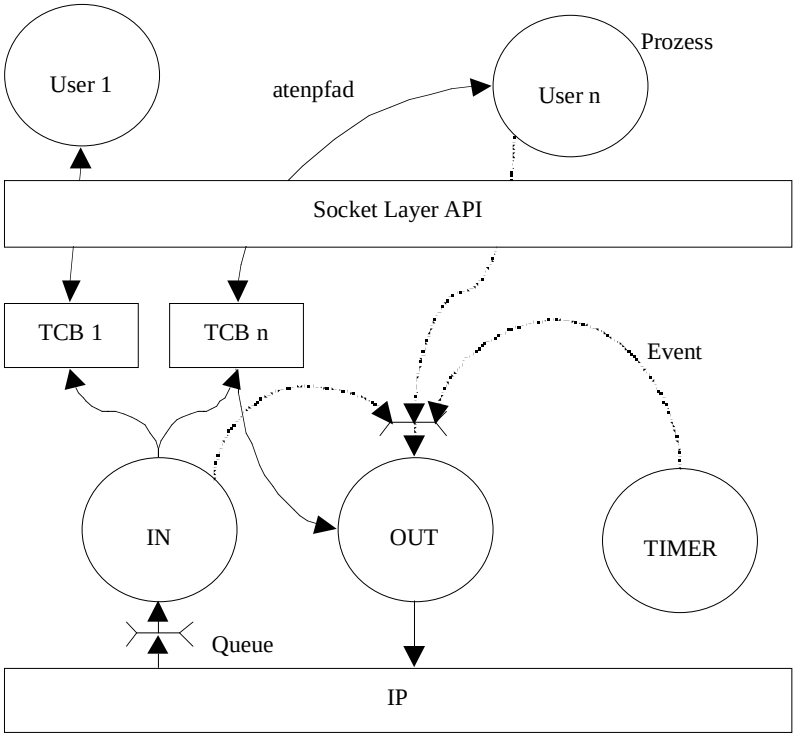

Abbildung 3.3-1, Thread-Modell

Vorteile einer Mehr-Prozess Implementation:

- TCP Piggybacking kann (einfacher) benutzt werden. Dadurch erhöht sich die Übertragungskapazität
- Schlankere und übersichtlichere Implementation

Nachteile einer Mehr-Prozess Implementation:

- mehr Kontext-Switchs
- es wird Code für Zugriff auf Datenstrukturen, welche von mehreren Threads benutzt werden, benötigt

3.3.2 Datenempfang

Alle Daten vom unterliegenden Layer werden an die Funktion *tcpInput* geliefert. Diese Funktion füllt die interne Datenstruktur *PacketRec* ab, welche in fast allen TCP Fuktionen verwendet wird.

Als erstes wird in *tcpInput* die interne Datenstruktur *PacketRec* abgefüllt. Danach wird geprüft, ob das empfangene Paket korrekt ist. Trifft das zu, wird anhand des TCP Ports der für diese Verbindung zuständige «Transmission Control Block», kurz TCB, mit *tcpDemux* gesucht. Danach wird mit *tcpOK* geprüft, ob das empfangene Paket zum aktuellen Verbindungsstatus passt. Mit dem Aufruf der Funktion *tcpInputDispatcher* wird schliesslich je nach Verbindungsstatus in die eigentliche Datenverarbeitungsfunktion verzweigt.

Das folgende Listing zeigt die *tcpInput* Funktion:

```
	void
tcpInput(				// does tcp input data processing
	NetBuf		buf,		// contains received data
	NetAttrs	attrs)		// contains some connection attributes
{
	TCBPtr		pTCB;		// represents one connection
	PacketRec	recPacket;	// internal structure for param passing
	int		err;

	recPacket.pTCPSeg = (TCPSegPtr)NETBUF_DATA(buf);
	recPacket.segBuf = buf;
	recPacket.netAttrs = attrs;
	err = TCP_NO_ERR;

	if (netbufLen(recPacket.segBuf) < sizeof(TCPSegRec))
		err = TCP_HEADER_TOO_SHORT_ERR;
	else if (netbufLen(recPacket.segBuf)
			< ntohs(TCP_HEADER_LEN(recPacket.pTCPSeg)))
		err = TCP_HEADER_TOO_SHORT_ERR;
	else
	{
		if (recPacket.pTCPSeg->u16CheckSum != 0)
		{
			if (tcpCheckSum(buf,attrs,netbufLen(buf)) != 0)
				err = TCP_CHECKSUM_ERR;
			else
			{
				// converts to local host byte order
				tcpHeaderNet2Host(recPacket.pTCPSeg);

				// searches the TCB block which maches current tcp port
				pTCB = tcpDemux(&recPacket);
				if (pTCB == nil)
				{
					// unexpected segement has arrived.
					tcpReset(&recPacket);
					err = TCP_NO_LISTENER_ERR;
				}
				else
				{
					if (tcpIsOK(pTCB,&recPacket))
					{
						tcpOptions(pTCB,&recPacket);
						tcpInputStateDispatcher(pTCB,&recPacket);
					}
					else
						tcpAckIt(pTCB,&recPacket);
					tmSemSignal(&pTCB->recMutex);
				}
			}
		}
	}
	netbufFree(buf);
	netattrFree(attrs);
}
```

3.3.3 Datenversand

Der Versand von Daten wird ausschliesslich vom Output-Thread getätigt. Will der Input-Thread beispielsweise eine Datenbestätigung senden, tut er dies nicht selbst, sondern teilt dem Output-Thread sein Begehren mit. Auch der Timer versendet keine Daten, sondern sendet dem Output-Thread ein Event, sobald z.B. ein Paket neu übermittelt werden muss.

Das folgende Listing zeigt die Hauptroutine des Output-Threads:

```
void tcpOut(ThreadArg arg)
{
	ThreadId	myThreadId,parentThreadId;
	TCBPtr		pTCB;
	unsigned long	u32Event;

	tmGetInfo(SELF,&myThreadId,&parentThreadId);
	netmodAdd(NETMODULE_TCP_OUT, myThreadId);
	ptInit(&gOutputPort,TCP_QUEUE_LEN);

	while (TRUE)
	{
		ptReceive(&gOutputPort,&pTCB,&u32Event);

		if (pTCB->setState <= TCP_STATE_CLOSED) continue;
		tmSemWait(&pTCB->recMutex);
		if (pTCB->setState <= TCP_STATE_CLOSED) continue;

		if (u32Event == DELETE)
			tcpTCBDeallocate(pTCB);
		else
		{
			tcpOutDispatcher(pTCB,u32Event);
			tmSemSignal(&pTCB->recMutex);
		}
	}
}
```

Der Output-Thread blockiert in der Funktion *ptReceive* bis er ein Event empfangen hat. Ist es ein «DELETE-Event» wird der aktuelle TCB freigegeben. Ansonsten wird mit *tcpOutDispatcher* in die dem aktuellem Status entsprechende Ausgabe-Funktion verzweigt.
Im Gegensatz zum Input-Thread, dessen Stati in den RFC beschrieben sind, hat der Output-Thread keine vorgeschrieben Stati. Diese Implementation verwendet vier verschiedene Stati um den Datenversand zu regeln. Es sind dies IDLE, TRANSMIT, RETRANSMIT und PERSIST. Die ersten drei Stati sind selbsterklärend, in den vierten gelangt der Output-Thread sobald das Empfangsfenster bei der Gegenstelle auf Null geschrumpft ist. Der Output-Thread prüft in diesem Status periodisch, ob die Fenstergrösse sich wieder vergrössert hat. Er wartet dabei, bis die Empfangsfenstergrösse auf ein Viertel der Initialgrösse gestiegen ist, um das «Silly-Window-Syndrom» [RFC793] zu verhindern.

3.3.4 Änderungen

Topsy verwendet für den Austausch von Daten zwischen den verschiedenen Layern NetBuf und NetAttrs [DWS]. Diese Eigenheit musste bei der Portierung des XINU TCP Codes berücksichtigt werden.

Beim Senden von einem TCP Paket muss daher nicht nur der TCP Header korrekt ausgefüllt werden, sondern zusätzlich muss eine NetAttr Datenstruktur alloziert und ausgefüllt werden. Das folgende Code Fragment zeigt die Attribute die gesetzt werden müssen.

```
netattrSet(attrs,NETATTR_TCP_FROM,u16SrcPort));
netattrSet(attrs,NETATTR_TCP_TO,u16DstPort));
netattrSet(attrs,NETATTR_IP_PROTOCOL,NETIP_PROTOCOL_TCP));
```

Die in Topsy verwendeten NetBuf haben die Eigenheit, dass die Daten nicht in einem Block ab einer bestimmten Speicheradresse verfügbar sind, sondern in mehrere einzelne kleinere Blöcke aufgeteilt sein können. Diese Eigenschaft muss in der Sende- (*tcpSend*), bzw. Empfangsfunktion (*tcpData*) berücksichtigt werden.
Für diesen Zweck wurde das NetBuf Modul um die beiden Funktionen *netbufRead* und *netbufWrite* erweitert.

In Gegensatz zu XINU TCP werden bei der Netzwerk Initialisierung nicht alle TCB gleich alloziert und in einem Array fixer Grösse verwaltet, sondern es wird jeder einzelne TCB dynamisch nach Bedarf aufgebaut. Die einzelnen TCB werden in einer verketteten Liste verwaltet. Auf den Kopf der Liste wird mit der globalen Variable *gTCPTCBList* gezeigt. Auf diese Liste wird von verschiedenen Threads zugegriffen. Gegenseitiger Auschluss wird daher über die Semaphore *gTCBLinkedListMutex* gewährleistet.

3.4 Timer

Topsy verfügt über zwei Clock-Interrupts (CLOCK0 und CLOCK1), die von einem Timerbaustein erzeugt werden. Bis anhin wurde in der Interruptserviceroutine von CLOCK0 nur der Scheduler aufgerufen. CLOCK1 wird zur Zeit nicht verwendet. Neu wird für den Timer der CLOCK0 verwendet. Der Scheduler wird nun auch vom Timer aufgerufen.

3.4.1 Übersicht

Der Timer verwaltet eine Queue, in der die Timer-Events mit ihren Daten eingetragen werden. In der Queue wird für die Elemente das Delta zum vorhergehenden Event eingetragen. Neue Events müssen daher in der Liste am richtigen Ort eingetragen werden. Ein Element in der Queue kann mehrere Argumente aufnehmen, die bei Ablauf des Timers zurückgeliefert werden. Weiter wird eine Kernel-Callbackroutine eingetragen, die beim Ablauf aufgerufen wird. Der User-Timer trägt in einem der Argumente ebenfalls eine User-Callbackroutine ein, die vom User-Timer-Thread aufgerufen wird, wenn dieser eine Timer-Expired-Message bekommt.

3.4.1.1 Delta-Timer

Der Timer benötigt etwas Zeit beim Einfügen einer Timer-Message in die Queue, dafür fast keine beim Testen, ob ein Timer abgelaufen ist. Der Timer besteht aus dem Kernel- und dem User-Teil. Im nachfolgenden Bild sind die verschiedenen Module dargestellt. Die Zeit des vordersten Elements wird bei jedem Interrupt dekrementiert. Danach wird für jedes Element, dessen Zeit auf 0 steht, die entsprechende Callbackroutine mit den Argumenten aufgerufen.

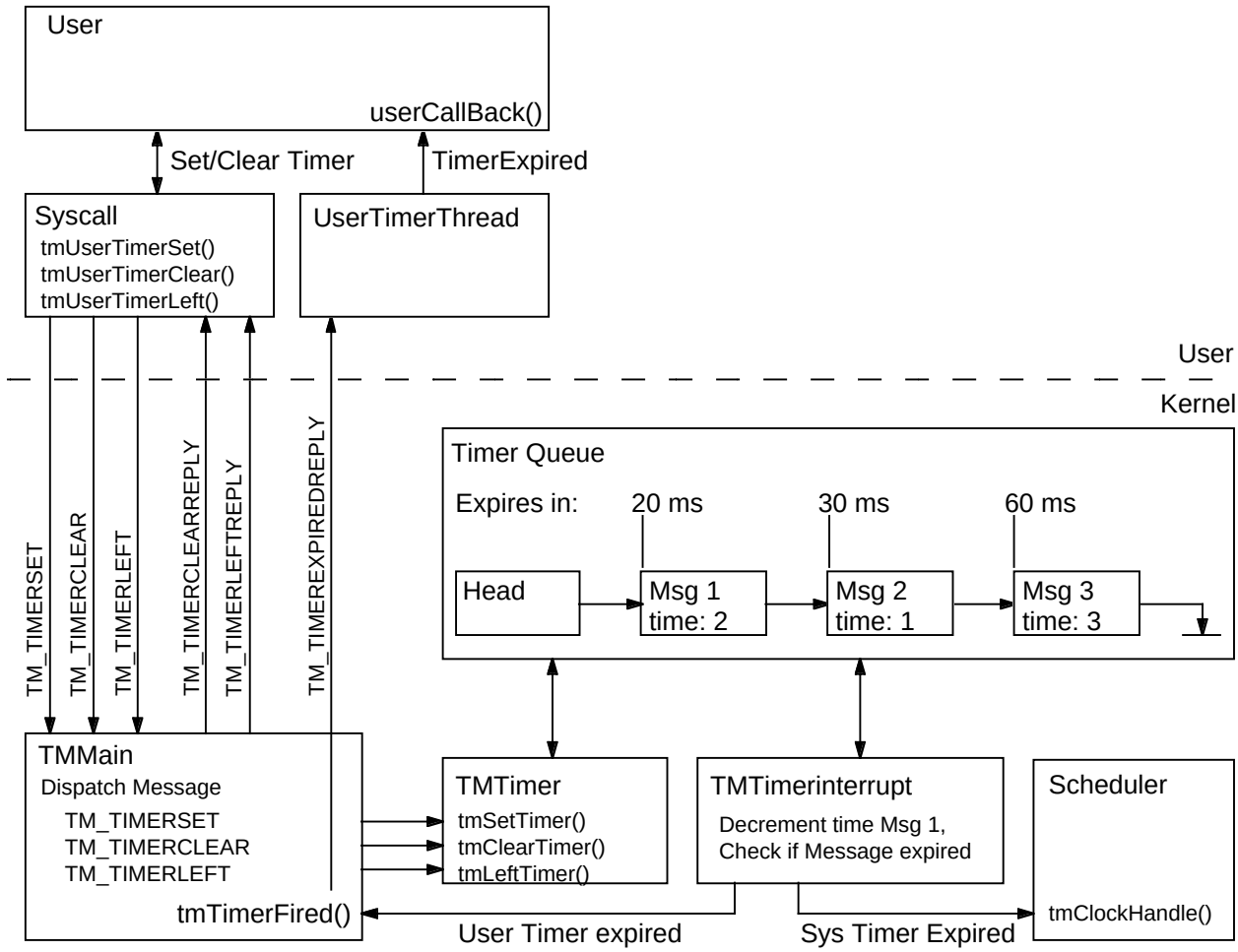

Abbildung 3.4-1, Delta-Timer

3.4.2 Kernel Timer

Im Thread-Manager Teil von Topsy wurde das Modul TMTimer eingefügt. Dieses verfügt über folgenden Funktionen:

3.4.2.1 tmTimerInit

```
void tmTimerInit();
```

tmTimerInit initialisiert die Timer-Queue auf nil sowie den Tick-Counter auf 0 und muss vor allen anderen Timer-Calls aufgerufen werden.

3.4.2.2 tmTimerInterrupt

```
void tmTimerInterrupt();
```

Interrupt-Service-Routine, die in TMInit für den CLOCK0 Interrupt gesetzt wird. Der CLOCK0 ist auf 10ms eingestellt.

3.4.2.3 tmGetTickCount

```
long int tmGetTickCount();
```

Liefert die Anzahl Ticks, die seit dem Systemstart vergangen sind.

3.4.2.4 tmSetTimer

```
      int                               //returns TIMER_OK or TIMER_FAILED
tmSetTimer(                             //sets timer
             tmCallBackProc fire,       //function is called if timer is expired
             void *pParam,              //user void * which is returned in fire
             long n32Param,             //user long which is returned in fire
             void *pMsg1,               //user void * which is returned in fire
             void *pMsg2,               //user void * which is returned in fire
             unsigned long u32Msg,      //user long which is returned in fire
             int time,                  //expiring time (time*10ms)
             Boolean bRepeat);          //if true, timer is periodic
```

Setzt eine Timer-Message in die Timer-Queue. Falls der Timer abläuft, wird die Callback-Procedure mit den entsprechenden Argumenten aufgerufen. Falls *bRepeat* gesetzt ist, läuft der Timer periodisch ab, ohne dass er neu gesetzt werden muss. Wird *tmSetTimer* mit Argumenten aufgerufen, die identisch mit einer existierenden Timer-Message sind, wird diese überschrieben.

3.4.2.5 tmClearTimer

```
      int                               //returns the remaining time
tmClearTimer(                           //clears the timer
             tmCallBackProc fire,       //used to compare with timer messages
             void *pParam,              //used to compare with timer messages
             long n32Param,             //used to compare with timer messages
             void *pMsg1,               //used to compare with timer messages
             void *pMsg2,               //used to compare with timer messages
             unsigned long u32Msg);     //used to compare with timer messages
```

Löscht eine Timer-Message aus der Timer-Queue, die mit den mitgegebenen Argumenten übereinstimmt. Als return value wird die verbleibende Zeit, bis der Timer abgelaufen wäre, in Anzahl Ticks (10ms) zurückgeliefert.

3.4.2.6 tmLeftTimer

```
      int                               //returns the remaining time
tmLeftTimer(                            //how much time is left
             tmCallBackProc fire,       //used to compare with timer messages
             void *pParam,              //used to compare with timer messages
             long n32Param,             //used to compare with timer messages
             void *pMsg1,               //used to compare with timer messages
             void *pMsg2,               //used to compare with timer messages
             unsigned long u32Msg);     //used to compare with timer messages
```

Als return value wird die verbleibende Zeit der Timer-Message, die mit den gelieferten Argumenten übereinstimmt, in Anzahl Ticks (10ms) zurückgeliefert.

3.4.2.7 tmCallBackProc

```
void (*tmCallBackProc)(    void *pParam,
                           long n32Param,
                           void *pMsg1,
                           void *pMsg2,
                           unsigned long u32Msg);
```

Bei Ablauf des Timers wird die in *tmSetTimer* mitgegebene Callback-Procedure mit den entsprechenden Argumenten aufgerufen.

3.4.3 Timer Init

In *tmInit* wird der System-Clock folgendermassen initialisiert:

```
tmTimerInit();
tmSetInterruptHandler(CLOCKINT_0,tmTimerInterrupt,NULL);
tmSetTimer(tmClockHandler,NULL,0L,NULL,NULL,0L,1,TRUE);
```

Dies entspricht einem periodischen Timer von 10ms. Die Funktion *tmTimerInterrupt* ruft alle 10ms *tmClockHandler* auf welche wiederum *schedule* aufruft.

3.4.4 User Timer

Um die Timer aus dem User-Space benützen zu können, wurde das Modul Syscall erweitert. Die Argumente korrespondieren mit den im Kernel Timer beschriebenen.

```
int tmUserTimerSet( tmUserTimerProc userCallBack,
                    void *pMsg1,
                    void *pMsg2,
                    unsigned long u32Msg,
                    int nTime,
                    Boolean bRepeat);

int tmUserTimerClear(tmUserTimerProc userCallBack,
                    void *pMsg1,
                    void *pMsg2,
                    unsigned long u32Msg);
```

```
int tmUserTimerLeft( tmUserTimerProc userCallBack,
                     void *pMsg1,
                     void *pMsg2,
                     unsigned long u32Msg);

long int tmUserTimerGetTicks();
```

3.4.5 User Timer Thread

Damit die Callback-Procedure eines abgelaufenen User-Timers aufgerufen werden, muss der User-Timer-Thread im Modul UserTimer gestartet werden.

```
int tmUserTimerInit()
```

3.4.6 Net Timer

Die Anpassung des User-Timers an die in TCP benötigten Timer wird im Modul *nettcpTimer* vorgenommen. Der einzige Unterschied zum User-Timer ist, dass die richtigen Typen für die in TCP benötigten Argumente benutzt werden.

3.5 Semaphoren

Für die Kommunikation sowie den exklusiven Ausschluss zwischen den TCP-Prozessen wird ein Semphore-Modul benötigt. Dieses verfügt über folgende Funktionen:

```
int tmSemInit(SemaphorePtr pSem,int nsCount);           //initialize Semaphre
int tmSemReset(SemaphorePtr pSem,int nsCount);          //reset Semaphre
int tmSemCount(SemaphorePtr pSem);                      //return sem count
int tmSemWait(SemaphorePtr pSem);                       //dec sem, wait if count < 0
int tmSemSignal(SemaphorePtr pSem);                     //inc sem
int tmSemSignalIfLess(SemaphorePtr pSem,int cmpValue);  //increments sem if count < n
```

tmSemWait dekrementiert die *count* Variable und ruft solange *tmYield* auf, bis die *count*-Variable der Semaphore >= 0 ist. *tmSemSignal* inkrementiert die *count* Variable, benachrichtigt den allfällig wartenden Prozess jedoch nicht. Dies könnte bei der Verwendung dieses Moduls für andere Zwecke zum Aushungern eines Prozesses führen. Bei der TCP-Implementation kann der Fall des Aushungerns nicht eintreten. Die Gefahr des Aushungerns kann vermindert werden, indem in *tmSemSignal* ein *tmYield* ausgeführt wird, wenn die count Variable auf 0 erhöht wurde.

3.6 Sockets

Die Socket Implementation in Topsy wurde so weit als möglich von XINU übernommen. In diesem Kapitel wird kurz erklärt wie die Implementation funktioniert und was bei der Portierung von XINU nach Topsy geändert werden musste.
Die eigentliche Socket Implementation konnte vollständig übernommen werden und ist in [DC2] ausführlich und komplett beschrieben. Deswegen wird in diesem Dokument nur teilweise darauf eingegangen.

Eine genaue Beschreibung der Anwendung der verschiedenen Funktionen ist im Kapitel «Benutzung des Socket-Layer» ausführlich beschrieben.

3.6.1 Änderungen

Im Gegensatz zu XINU gibt diese Implementation als Socket nicht ein Integer zurück, sondern eine Adresse auf eine Socket-Datenstruktur.

Die Sende- bzw. Empfangsroutinen in XINU sind nur blockierend, wenn die Grösse der mitgegebenen Buffer bzw. Längen die der internen TCP Buffer nicht übersteigen. Diese Unschönheit ist mit den folgenden beiden Funktionen nun nicht mehr der Fall:

```
      int                              // returns num of sent bytes or error (<0)
tcpWrite(                              // transmits data
      SocketPtr    pSocket,            // socket ptr
      unsigned char*pu8SrcBuf,         // source data buffer
      unsigned longu32Len,             // length of source data buffer
      Boolean      bIsUrgent)          // if TRUE, urgent data processing will be used
{
      long int nResult = 0;
      long int nBytesWritten = 0;

      while (nBytesWritten<u32Len)
      {
            // are remaining data > TCP_SND_BUF_SIZE
            if (u32Len-nBytesWritten>TCP_SND_BUF_SIZE)
                  nResult = tcpWriteBlock(pSocket,pu8SrcBuf+nBytesWritten,
                                          TCP_SND_BUF_SIZE,bIsUrgent);
            else
                  nResult = tcpWriteBlock(pSocket,pu8SrcBuf+nBytesWritten,
                                              u32Len-nBytesWritten,bIsUrgent);
            if (nResult<0L)
                  return nResult;
            else
                  nBytesWritten+=nResult;

            if ((pSocket->pTCB->u16Flags & TCP_TCB_BUFFER) == 0)
                  break;
            else if (nBytesWritten<u32Len)
                  tmYield();
      };
      return nBytesWritten;
}
```

```
	int                             // num of bytes read or status/error code
tcpRead(                            // reads data
	SocketPtr    pSocket,           // socket ptr
	unsigned char*pu8SrcBuf,        // source data buffer
	unsigned longu32Len)            // length of source data buffer
{
	long int nResult = 0;
	long int nBytesRead = 0;

	while (nBytesRead<u32Len)
	{
		// are remaining data to read > TCP_RCV_BUF_SIZE
		if (u32Len-nBytesRead>TCP_RCV_BUF_SIZE)
			nResult = tcpReadBlock(pSocket,pu8SrcBuf+nBytesRead,
                                                  TCP_RCV_BUF_SIZE);
		else
			nResult = tcpReadBlock(pSocket,pu8SrcBuf+nBytesRead,
                                                 u32Len-nBytesRead);
		if (nResult<0L)
			return nResult;
		else
			nBytesRead+=nResult;

		if ((pSocket->pTCB->u16Flags & TCP_TCB_BUFFER) == 0)
			break;
		else if (nBytesRead<u32Len)
			tmYield();
	};
	return nBytesRead;
}
```

Natürlich sind beide Funktionen nur blockierend, wenn die Benutzer-Option *TCP_TCB_BUFFER* mit *tcpControl* auch gesetzt wurde.
Die Sende- und Empfangsroutinen von XINU wurden nicht mehr in das Socket Layer API aufgenommen.

3.7 Änderungen in Topsy

In folgenden Modulen wurden Änderungen vorgenommen:

3.7.1 Messages

Die Messages Include Datei wurde um die Timer Event/Resultat Codes erweitert. Weiter wurde die Message-Datenstruktur um die benötigten Timer-Strukturen ergänzt. Zudem musste die Message-Datenstruktur vergrössert werden.

3.7.2 NetModules

Die Grösse der NetModul eigenen Message-Datenstruktur wurde der in Messages verwendeten angepasst.

3.7.3 Syscall

Im Syscall Modul wurden sämtliche User-Timer Aufrufe implementiert. Die Beschreibung des User-Timer findet sich im gleichnamigen Kapitel.

3.7.4 TMInit

Im Modul TMInit wird neu der Timer initialisiert und die benötigte Interrupt-Routine gesetzt. Die war früher im Modul Error. Zudem wird ein periodischer Timer für den Scheduler gesetzt. Dieser wurde früher direkt in der Timer-Interrupt-Service-Routine aufgerufen.

3.7.5 TMMain

Das TMMain Modul wurde um die Timer-Messages erweitert.

3.7.6 Shell

Zwei neue Programme wurden in der Shell installiert. Es sind dies die Applikationen «httpd» und «echod».

3.7.7 NetBuf

NetBuf wurde um zwei Funktionen erweitert, um bequem in NetBufs zu schreiben und von diesen Daten zu lesen.

3.7.7.1 netbufWrite

```
long netbufWrite(NetBuf pDest,unsigned char *pSrc,long n32Len,long n32DestOffset);
```

netbufWrite schreibt Daten von einem linearen Buffer in einen NetBuf. Der NetBuf muss alloziert sein und mindestens über freien Platz von *n32Len+n32DestOffset* verfügen. *n32DestOffset* gibt an, wieviele Bytes im NetBuf vor den kopierten Daten freigelassen werden.

3.7.7.2 netbufRead

```
long netbufRead(unsigned char *pDest,NetBuf pSrc,long n32Len,long n32SrcOffset);
```

netbufRead liest Daten aus einem NetBuf in einen linearen Buffer. Der Buffer muss alloziert sein und über die Länge *n32Len* verfügen. *n32SrcOffset* gibt an, ab welcher Position die Daten ausgelesen werden sollen.

3.7.8 NetConfig

Ergänzung der Informationen für das TCP-Modul.

4 Benutzung des Socket-Layers

4.1 Funktionsbeschreibungen

Der Socket Layer stellt eine Reihe von Funktionen einer Benutzer Applikation zur Verfügung, die eine Kommunikation über TCP wesentlich erleichtern.

Alle Funktionen retournieren im Fehlerfall einen negativen Error-Code. Die Bedeutung der verschiedenen Codes sind in der folgenden Tabelle aufgelistet:

Code	**Bedeutung**
TCP_NO_ERR	Funktion konnte ordnungsgemäss ausgeführt werden
TCP_GENERAL_ERR	Allgemeiner Fehler; keine nähere Spezifikation
TCP_RESET	Verbindung wurde abbgebrochen
TCP_REFUSED	Verbindungsaufnahme wurde verweigert
TCP_TIMEDOUT	Daten wurden auch nach mehrmaliger Wiederholung von der Gegenstelle nicht bestätigt

Tabelle 4.1-1, Error-Codes

Im Folgenden sind die verfügbaren Funktionen im Detail beschrieben.

4.1.1 tcpOpen

Mit *tcpOpen* wird eine passive oder aktive Verbindung eröffnet. «Passive Open» werden von Server Applikationen benötigt. Sie erlauben ein «Listen» auf eine Verbindung. Mit «aktiven Open» eröffnet ein Client eine Verbindung mit einem Server.

```
      int                               // returns TCP_NO_ERR if successful
tcpOpen(                                // passive or active open
      IPAddressPtr pIPSrcAddr,          // source ip addrs
      IPAddressPtr pIPDstAddr,          // dest ip addrs
      unsigned short u16SrcPort,        // local tcp port or TCP_ANY_LOCAL_PORT
      unsigned short u16DstPort,        // TCP_ANY_FOREIGN_PORT = Server
      SocketPtr    *ppSocket);          // returns allocated socket
```

Wenn ein passives Öffnen einer Verbindung vollzogen werden soll, muss als Source-Port ein gültiger TCP Port mitgegeben werden, auf dem der Server horcht. Als Dest-Port muss die Konstante *TCP_ANY_FOREIGN_PORT* mitgegeben werden.
Bei einem aktiven Öffnen einer Verbindung muss als Source-Port die Konstante *TCP_ANY_LOCAL_PORT* und als Dest-Port die Server Port Nummer angegeben werden.
Die Funktion retourniert einen negativen Fehler-Code oder *TCP_NO_ERR*.

4.1.2 tcpClose

Mit *tcpClose* wird eine Verbindung geschlossen und der allozierte Socket freigegeben. Die Funktion retourniert einen negativen Fehler-Code oder *TCP_NO_ERR*.

```
      int                              // returns TCP_NO_ERR if successful
tcpClose(                              // closes a connection. dispsoeses socket
      SocketPtr pSocket);              // socket
```

4.1.3 tcpControl

Mit *tcpControl* können verschiedenen Optionen einer Verbindung gesteuert werden. Die Funktion retourniert einen negativen Fehler-Code oder *TCP_NO_ERR*.

```
      int                              // Error code, or TCP_NO_ERR
tcpControl(                            // sets various socket options
      SocketPtr    pSocket,            // socket
      int          nOperation,         // control operation
      void         *arg,               // argument 1 of operation
      void         *arg2,              // argument 2 of operation
      void         **pResult);         // returns a result
```

Die folgende Liste zeigt die möglichen Operationen und deren Funktion:

Operation	**Arg**	**Arg2**	**ppResult**	**Bedeutung/Funktion**
TCP_CONTROL_ LISTEN_QUEUE	Listen Queue Size			Setzt die maximale Anzahl der Clients, die sich mit einem Server verbinden können.
TCP_CONTROL_ ACCEPT			neuer Socket	Wartet, bis ein Client sich mit dem Server verbindet. In der Variable *ppResult* wird der neue Socket zurückgeliefert.
TCP_CONTROL_ STATUS	Pointer auf Statistik Datenstruktur			Gibt eine Statistik über die aktuelle Verbindung aus.
TCP_CONTROL_SET_ USER_OPTION	Benutzer Option			Setzt eine Benutzer-Option
TCP_CONTROL_CLR_ USER_OPTION	Benutzer Option			Löscht eine Benutzer-Option
TCP_CONTROL_SEND_ URGENT	Pointer auf «urgent» Daten Buffer	Länge der «urgent» Daten		Sendet «urgent» Daten

Tabelle 4.1-2, Control-Operationen

Die folgende Liste zeigt die möglichen Benutzer Optionen und deren Funktion:

Option	**Bedeutung/Funktion**
TCP_TCB_DELACK	Wenn diese Option gesetzt wird, werden TCP Acknowledge verzögert. Dadurch wird Piggybacking besser ausgenutzt.
TCP_TCB_BUFFER	Wenn diese Option gesetzt wird blocken die *tcpRead* und *tcpWrite* Funktionen

Tabelle 4.1-3, Benutzer-Optionen

4.1.4 tcpRead

Mit *tcpRead* können Daten von der Gegenstelle empfangen werden. Die Funktion verhält sich je nach Benutzer Option synchron oder asynchron.

```
      int                              // num of bytes read or error/status code
tcpRead(                               // reads urgent/normal data from TCP buffers
      SocketPtr    pSocket,            // socket
      unsigned char*pu8SrcBuf,         // user buffer
      unsigned longu32Len);            // size of buffer
```

Sobald «urgent» Daten empfangen werden, retourniert *tcpRead* den Status-Code *TCP_URGENTMODE*. Die Benutzer Applikation weiss nun, dass «urgent» Daten darauf warten, konsumiert zu werden. Sobald das Ende der «urgent» Daten erreicht ist, wird der Status-Code *TCP_NORMALMODE* zurück gegeben. Dieses Verhalten ist unabhängig davon, ob *tcpRead* synchron oder asynchron aufgerufen wurde.
Ist der zurückgelieferte Wert positiv, gibt dieser die Anzahl der gelesenen Bytes an. Ist der Code negativ, aber nicht gleich *TCP_URGENTMODE* oder *TCP_NORMALMODE*, so ist ein Fehler aufgetreten.

4.1.5 tcpWrite

Mit tcpWrite können Daten an eine entfernte Applikation gesendet werden. Der Parameter *bUrgent* gibt an, ob die Daten im «urgent» Mode übertragen werden sollen. Die Funktion verhält sich je nach Benutzer Option synchron oder asynchron.

```
      int                              // returns number of sent bytes or error (<0)
tcpWrite(                              // transmits data
      SocketPtr    pSocket,            // socket ptr
      unsigned char*pu8SrcBuf,         // source data buffer
      unsigned longu32Len,             // length of source data buffer
      Boolean      bIsUrgent);         // if true, urgent data processing will be used
```

Ist der zurückgelieferte Wert positiv, gibt dieser die Anzahl der geschriebenen Bytes an. Ist der Code negativ, so ist ein Fehler aufgetreten.

4.2 Beispiel

Der folgende Abschnitt zeigt, wie mit dem oben beschriebenen Socket Layer ein einfacher Server, bzw. Client realisiert werden kann. Der in den folgenden Beispielen gezeigte Code verfügt nur über die absolut notwendigen Fehlerbehandlungen.

4.2.1 Server

Das folgende Beispiel zeigt den Code eines Echo Servers. Der Code besteht im Wesentlichen aus zwei Teilen; ein Teil kümmert sich um die eigentliche Datenkommunikation mit einem Client, der andere Teil ist verantwortlich für die Verbindungsaufnahme eines Clients mit einem Server.

Der Server alloziert für jede neue Verbindung mit einer Gegenstelle einen eigenen Thread, der sich danach um die Kommunikation kümmert. Die Datenübertragung wird synchron und ohne «urgent» Daten Behandlung vollzogen.

```
// echo Thread
void tcpEchoProcess(ThreadArg arg)
{
	char		buf[TCP_ECHO_BUFFER_LEN];
	char		cChar;
	int		nResult;
	SocketPtr	pSocket = (SocketPtr)arg;

	while (TRUE)
	{
		nResult = tcpRead(pSocket,&cChar,1);
		if (nResult == TCP_URGENTMODE || nResult == TCP_NORMALMODE) continue;
		if (nResult <= 0) break;
		nResult = tcpWrite(pSocket,&cChar,1,FALSE);
		if (nResult <= 0) break;
	}
	nResult = tcpClose(pSocket);
}

// echo server
void tcpEchoDeamon(ThreadArg arg)
{
	IPAddressUon	uonSrcAddr,uonDstAddr;
	int		nError;
	SocketPtr	pSvrSkt,pSkt;
	ThreadId	echoID;
	void		*pTmp;
	char		cChar;

	uonSrcAddr.u32Addr = 0xC0A80202;	// 192.168.2.2
	uonDstAddr.u32Addr = 0xC0A80201;	// 192.168.2.1

	nError = tcpOpen(&uonSrcAddr,&uonDstAddr,7,TCP_ANY_FOREIGN_PORT,&pSvrSkt);
	if (nError == TCP_NO_ERR)
	{
		nError = tcpControl(pSvrSkt,TCP_CONTROL_LISTEN_QUEUE,10,0,&pTmp);
		if (nError == TCP_NO_ERR)
		{
			while (TRUE)
			{
```

```
                        nError = tcpControl(pSvrSkt,TCP_CONTROL_ACCEPT,0,0,&pSkt);
                        if ((nError != TCP_NO_ERR) && (pSkt!= nil)) break;
                        tcpControl(pSkt,TCP_CONTROL_SET_USER_OPTION,
                                                TCP_TCB_BUFFER,0,0);
                        tmStart(&echoID,tcpEchoProcess,pSkt,"tcpEchoDeamon");
                }
        }
    }
}
```

4.2.2 Client

Der folgende Code zeigt einen einfachen Echo Client. Das Programm sendet eine gewisse Anzahl Bytes an die Gegenstelle und wartet, bis diese wieder empfangen werden.

```
int tcpEchoClient(unsigned char *pu8Data,unsigned char *pu8Len)
{
    IPAddressUon uonSrcAddr,uonDstAddr;
    int          nResult;
    SocketPtr    pSkt;

    uonSrcAddr.u32Addr = 0xC0A80102;        // 192.168.1.2
    uonDstAddr.u32Addr = 0xC0A80101;        // 192.168.1.1

    nResult = tcpOpen(&uonSrcAddr,&uonDstAddr,TCP_ANY_LOCAL_PORT,7,&pSkt);
    if (nResult == TCP_NO_ERR)
    {
        nResult = tcpWriteBlock(pSocket,pu8Data,*pu8Len,FALSE);
        if (nResult <= 0) return nResult;

        nResult = tcpReadBlock(pSocket,pu8Data,*pu8Len);
        if (nResult <= 0) return nResult;

        nResult = tcpClose(pSocket);
    }
    return nResult ;
}
```

5 Inbetriebnahme/Systemtest

5.1 Testumgebung auf Macintosh

Zur einfacheren Portierung von TCP wurde diese mit dem Metrowerks-Compiler auf einem Macintosh ausgeführt. Aufgrund der komfortableren Debug-Möglichkeiten wurde eine Umgebung auf dem Macintosh aufgebaut, mit der die Funktionen des portierten TCP-Layers genau getestet werden konnten. Dazu mussten verschiedene Topsy-Module zumindest teilweise übernommen werden.

5.1.1 Portierte Topsy-Module

Lock, NetAttrNetBuf, NetConfig, NetDebug, NetInit, NetIP, NetIPchecksum, NetModules, NetSyscall, SupportSyscall

5.1.2 IP-Testmodul

Für die Simulation wurde ein spezielles NetIP-Modul implementiert, welches gegen TCP das identische Interface wie das Topsy NetIP-Modul hat. Es wurden verschiedene Simulationsmöglichkeiten eingebaut.

5.1.2.1 IP Paketsimulation

Mit der Simulationsengine wird ein möglichst realer, fehlerbehafteter IP-Kanal simuliert. Die Rate der Paketmanipulationen ist einstellbar. Folgende Manipulationen können in beliebigen Kombinationen eingeschaltet werden.

- Pakete verlieren
  Verwirft einzelne Pakete
- Pakete vertauschen
  Zwei aufeinanderfolgende Pakete werden vertauscht gesendet.
- Pakete duplizieren
  Sendet ein Paket identisch ein zweites Mal
- Pakete verzögern
  Sendet das Paket erst nach dem nächsten, das gesendet werden soll.
- Pakete zerstören
  Ein zufälliges Byte im Paket wird überschrieben. Dies kann im Header- oder im Datenteil geschehen

5.1.2.2 Test über Loopback

In einem ersten Schritt wurde jedes gesendete Paket nach dem Durchlaufen der Simulationsengine direkt wieder zum TCP-Layer hochgeschickt. Nach der Behebung einiger Fehler, funktionierte Connect sowie Disconnect einwandfrei und es konnten Dateien mit mehreren hundert Kilobytes auch bei eingeschalteter IP-Simulation fehlerfrei übertragen werden.

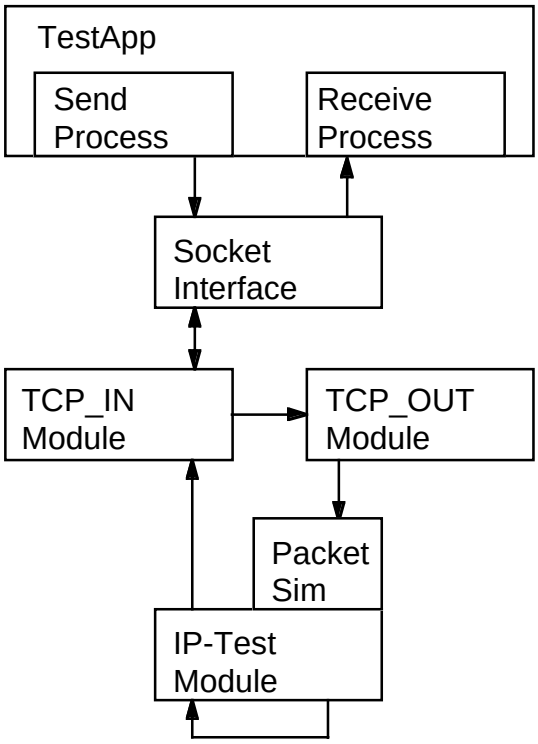

Abbildung 5.1-1, Loopback

5.1.2.3 Test über Raw IP Layer von OpenTransport

Pakete werden über einen Raw-IP Layer von OpenTransport auf dem Macintosh übers Netzwerk zu einer anderen Instanz gesendet. Die IP-Simulationsengine kann ebenfalls zugeschaltet werden.
Das ursprüngliche Ziel dieser Anordnung war, die TCP/IP Implementation gegen einen bestehenden TCP/IP Stack auszutesten. Dies war leider nicht möglich, da RawIP von OpenTransport aus Performance-Gründen keine TCP- und UDP-Pakete empfangen kann (Protocolfield im IP-Header auf 6 (TCP) oder 17 (UDP) gesetzt). Der Test zwischen zwei Instanzen dieser Implementierung mit einer anderen Protokollnummer verlief problemlos. Die Richtigkeit der TCP/IP-Implementation durch Kommunikation mit einem bestehenden TCP/IP-Stack muss daher auf Topsy nachgewiesen werden.

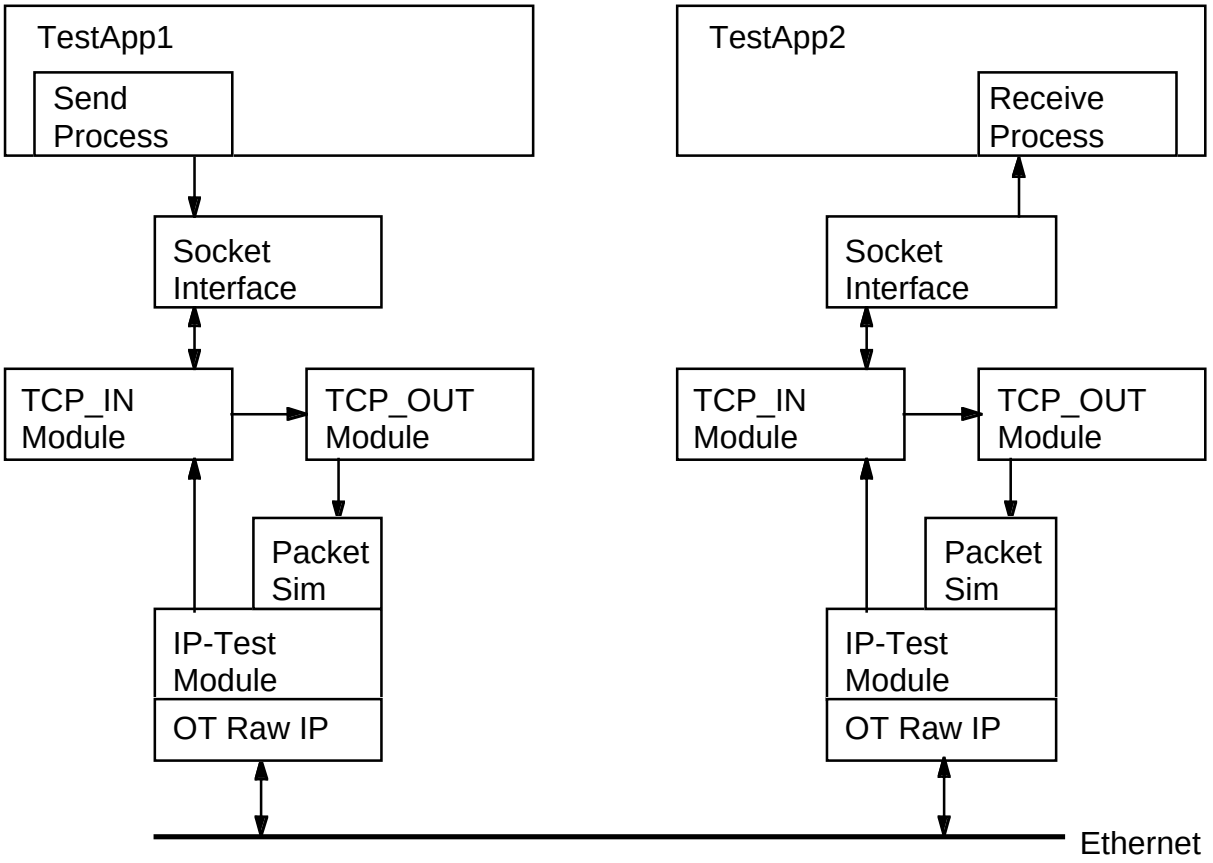

Abbildung 5.1-2, Raw IP Layer

5.2 Inbetriebnahme auf Topsy

Die auf MacOS getestete Implementation musste nun in Topsy übernommen werden. Dabei tauchten noch ein paar Probleme auf.

5.2.1 Probleme

Es mussten zwei Fehler behoben werden, die in der kooperativen MacOS-Umgebung nicht aufgetreten sind.
Die Hardwareadresse wurde im Ethernet-Modul byteweise verkehrt ins Ethernet-Paket geschrieben. Interessanterweise funktionierte ein Ping von Linux aus trotzdem, die TCP-Pakete kamen hingegen beim Linux-Socket nicht an.

5.2.2 Hilfsmittel

Zur Untersuchung der Netzwerkpakete wurden tcpDump und Ethereal (http://ethereal.zing.org) eingesetzt.

5.2.3 TCP-Echo

Der Echo-Deamon kann auf Topsy mit «start echod» gestartet werden. Der Deamon wartet auf eingehende Verbindungen auf Port 7. Er kann mit Telnet von Linux aus folgendermassen getestet werden:

telnet 192.168.2.2 7

Der Echo-Deamon schickt nach der Verbindungsaufnahme alle Daten die er empfängt, unverändert zurück.

5.2.4 Einfacher HTTP-Server

Der HTTP-Server kann auf Topsy mit «start httpd» gestartet werden. Der Deamon wartet auf eingehende Verbindungen auf Port 80. Bekommt er einen HTTP-Request, liefert er ein Webseite zurück. Die Webseite verfügt über einen Link auf eine andere Seite, einen Link auf einen Dateidownload sowie ein Editfeld und einen Button.

5.3 Performance-Test

5.3.1 Testumgebung

Für den Performance Test wurde das gleiche Programm wie in [DWS], Anhang C.4 beschrieben ist, verwendet.
Der Test wurde mit dem Echo Server gemacht. Mit einem Telnet Client wurden rund 40 Pakete à durchschnittlich 50 Bytes gesendet.

5.3.2 Testergebnisse

Das Testergebnis mit diesem einfachen Test kann nicht allzu hoch bewertet werden. Es sagt beispielsweise nichts darüber aus, wie schnell der TCP/IP Stack auf Topsy im Vergleich mit TCP/IP Stacks von anderen Betriebssystemen ist. Dazu müsste Topsy auf der Ziel-Hardware mit einem echten Ethernet Adapter betrieben werden. Trotzdem gibt die folgende Tabelle Aufschluss über einige interessante Werte wie Anzahl der Aufrufe einer bestimmten Funktion und der von ihr benötigten Anzahl Instruktionen.

Function	Aufrufe	Instruktion pro Aufruf	Instruktion x Aufrufe	Anteil in %
schedule	17810	74	1326303	9.65
saveContext	19030	52	1008549	7.34
restoreContext	19030	47	894410	6.51
hashListGet	23779	33	788609	5.74
msgDispatcher	17495	41	730696	5.31
tmMsgRecv	1945	364	708932	5.16
longCopy	4543	121	551245	4.01
listMoveToEnd	15865	30	490523	3.57
testAndSet	40616	11	487372	3.54
testAndSet	37572	11	420923	3.06
syscallExceptionHandler	17495	22	386572	2.81
listGetFirst	33196	11	383022	2.79
netattrGet	10096	35	356868	2.6
lock	22598	14	331103	2.41
zeroOut	224	1284	287631	2.09
lockTry	35616	8	284928	2.07
tmSemWait	6056	42	256583	1.87
hashFunction	23835	9	214515	1.56
getMessageFromQueue	3135	63	199682	1.45
kSend	3117	55	172026	1.25
tmMsgSend	13302	11	159608	1.16
findFree	214	671	143624	1.04
listSwap	3875	35	135625	0.99
netattrSet	4481	30	134442	0.98
tmYield	11379	10	125162	0.91
tmSemSignal	6055	17	102940	0.75
threadYield	11383	9	102447	0.75
tmTimerInterrupt	1383	69	96405	0.7
addMessageInQueue	1200	74	88941	0.65
netattrNew	273	314	85724	0.62
unlock	40609	1	81215	0.59
netbufAlloc	549	144	79498	0.58
netipChecksum	538	144	77701	0.57
netdbgPrintf	5485	13	71316	0.52
schedulerSetReady	1940	36	69840	0.51
schedulerSetBlocked	1935	36	69660	0.51
kRecv	3135	21	68040	0.49
tcpGetIPAddr	1352	46	62870	0.46
lock	5000	12	60000	0.44
removeElem	3875	15	59456	0.43
wordCopy	271	204	55356	0.4
intDispatcher	1534	34	53170	0.39

schedulerRunning	11439	4	45756	0.33
unlock	22605	1	45203	0.33
tcpWriteBlock	515	75	38893	0.28
netbufFreeBuf	540	67	36181	0.26
ipcResetPendingFlag	17800	2	35600	0.26
tmSetReturnValue	17495	2	34990	0.25
tmResetClockInterrupt	1534	22	34829	0.25
tmLeftTimer	455	70	32214	0.23
tcpWakeup	1035	30	31558	0.23
tcpReadBlock	516	58	30403	0.22
netethUp	138	213	29479	0.21
tmMain	1	28930	28930	0.21
tmSemSignalIfLess	1161	24	28635	0.21
tcpSend	132	211	27953	0.2
tcpGetData	515	53	27806	0.2
ptSend	464	55	25662	0.19
ptReceive	466	51	24162	0.18
tmMsgRecv	1190	19	23736	0.17
tcpTransmit	463	48	22481	0.16
hwExceptionHandler	1534	14	21476	0.16
tmInsertElement	1501	13	20184	0.15
netethDown	133	144	19197	0.14
tcpGetSpace	515	36	18545	0.13
tmClearTimer	258	70	18181	0.13
genericSyscall	982	18	17677	0.13
hmFree	27	625	16875	0.12
ioDelayAtLeastCycles	492	34	16730	0.12
netdbgDisplay	1655	10	16558	0.12
netcfgSendNextUp	413	39	16108	0.12
netifGetAttr	1218	13	15834	0.12
netcfgSendNextDown	398	38	15132	0.11
byteCopy	48	314	15116	0.11
tcpOut	1	13922	13922	0.1
netbufLen	1356	10	13560	0.1
tcpDemux	137	98	13439	0.1
netmodMsgRecv	956	13	13349	0.1
netmodSendUpThread	413	32	13217	0.1
tmMsgSend	1059	12	12709	0.09
tcpEchod	1	12598	12598	0.09
tcpKick	463	27	12502	0.09
ioDeviceMain	5	2496	12481	0.09
tcpHowMuch	725	17	12328	0.09
netmodSendDownThread	398	30	11940	0.09
tcpInput	137	85	11647	0.08
memCopy	271	42	11524	0.08
ioCheckBufferAddress	358	32	11457	0.08
tmUserTimerLeft	455	25	11376	0.08
netipForUs	137	82	11234	0.08
netcfgNextUp	413	26	11150	0.08
ptCount	464	24	11137	0.08
tcpData	137	80	11065	0.08
tmClockHandler	1374	8	10992	0.08
tcpCheckSum	269	40	10762	0.08
netipFillHeader	132	80	10563	0.08
tmSemCount	2089	5	10447	0.08

netmodSendUp	413	25	10325	0.08
netipDownSend	132	77	10166	0.07
tcpSetIPAddr	268	36	9782	0.07
netcfgarpResolveETHIP	132	74	9768	0.07
netmodSendDown	398	24	9553	0.07
tcpAcked	136	69	9440	0.07
netipUp	137	64	8768	0.06
netcfgNextDown	398	22	8758	0.06
tcpOutDispatcher	463	17	8176	0.06
hmAlloc	214	37	7919	0.06
tcpEchoProcess	1	7739	7739	0.06
tcp_tmLeft	455	17	7736	0.06
netbufFree	540	14	7560	0.05
tcpRoundTripTime	131	56	7452	0.05
netbufWrite	130	57	7410	0.05
netcfgarpLookupETHIP	133	53	7102	0.05
netcfgTransformDown	398	17	7024	0.05
tmSetTimer	127	54	6859	0.05
tcpIsOK	137	49	6721	0.05
nettapRead	138	48	6624	0.05
empty	0	-	6136	0.04
ethertap_write	133	46	6118	0.04
netipVerify	137	44	6031	0.04
netipDownSendFragment	132	44	5808	0.04
ioConsolePutChar	271	21	5747	0.04
tcpSndWindow	135	42	5729	0.04
netbufAddHead	265	20	5300	0.04
tcpEstablished	135	37	4995	0.04
ioRead	207	24	4968	0.04
byteCopy	51	95	4894	0.04
tcpDoData	137	34	4734	0.03
nettapMain	1	4652	4652	0.03
netethMain	1	4484	4484	0.03
memCopy	135	33	4474	0.03
ethertap_read	138	32	4416	0.03
netipSetAttrs	137	32	4384	0.03
devSCN2681_write	18	230	4145	0.03
netipMain	1	4053	4053	0.03
netudpInit	1	4022	4022	0.03
tcpOptions	137	28	3972	0.03
ethertap_interruptHandler	138	28	3864	0.03
listAddInFront	108	35	3839	0.03
tcpRcvWindow	132	28	3804	0.03
mmAddressSpaceRange	374	9	3729	0.03
initMsgQueue	23	162	3727	0.03
nettapDown	133	28	3726	0.03
netioWrite	133	28	3724	0.03
tcpSendLen	132	27	3693	0.03
mmMovePage	224	16	3584	0.03
tmUserTimerSet	126	28	3528	0.03
ioWriteFlags	133	25	3325	0.02
tcpInputStateDispatcher	137	24	3289	0.02
tmUserTimerClear	131	25	3275	0.02
netipDown	132	24	3169	0.02
split	214	14	3048	0.02

mmVmMove	14	209	2926	0.02
threadBuild	23	126	2916	0.02
nettcpMain	1	2626	2626	0.02
ioConsolePutString	13	198	2581	0.02
netcfgNextUpEth	138	18	2486	0.02
netcfgNextUpIP	137	18	2466	0.02
threadStart	21	113	2387	0.02
tcpOutputState	131	17	2347	0.02
tcp_tmClear	131	17	2232	0.02
vmAllocRegion	38	49	1870	0.01
stringNCopy	189	9	1844	0.01
mmVmAlloc	38	47	1786	0.01
tcp_tmSet	126	13	1638	0.01
mmMain	1	1439	1439	0.01
getRegion	28	50	1401	0.01
hashListAdd	28	43	1204	0.01
listRemove	27	41	1107	0.01
nettcpUp	137	8	1096	0.01
devSCN2681_read	69	14	975	0.01
getThreadId	21	45	959	0.01
netcfgTransformUp	413	2	826	0.01
tcpPacketDump	269	3	809	0.01
readChar	12	66	792	0.01
tcpIdle	79	10	790	0.01
genericSyscall	39	18	702	0.01
schedulerInsert	23	30	690	0.01
mmInitMemoryMapping	1	616	616	0
longCopy	51	11	591	0
wordCopy	33	17	571	0
vmFreeRegion	14	37	518	0
setTLBEntry	64	8	512	0
readLine	1	499	499	0
mmVmFree	14	33	462	0
vmAlloc	20	23	460	0
ioWrite	18	25	450	0
devSCN2681_interruptHandler	12	37	447	0
pqInitq	1	410	410	0
vmInitRegion	10	36	360	0
tmStart	13	27	351	0
vmMove	14	25	350	0
stringCompare	11	30	340	0
vsprintf	1	326	326	0
tmSetExceptionHandler	46	7	322	0
threadInfo	8	38	304	0
netarpUpETHIP	1	302	302	0
stringLength	2	150	300	0
tmSemInit	15	20	300	0
tmTimerInit	1	285	285	0
ptInit	3	94	283	0
roundToPageUp	55	5	275	0
tcpHeaderNet2Host	137	2	274	0
initBasicExceptions	1	273	273	0
start	1	266	266	0
netipRoute	132	2	264	0
tcpHeaderHost2Net	132	2	264	0

tmSetInterruptHandler	13	19	258	0
netbufRead	5	48	240	0
tmGetInfo	8	29	232	0
isPageInRegion	28	8	224	0
devSCN2681_init	4	54	219	0
ioMain	1	216	216	0
listNew	11	18	198	0
netMain	1	177	177	0
hashListNew	2	84	168	0
mmVmInit	1	162	162	0
ioLoadDriver	5	30	154	0
tcpFragmentJoin	5	29	145	0
netmodAdd	8	18	144	0
tcpControl	3	46	140	0
ioConsolePutInt	2	68	136	0
tmStart	5	27	135	0
netbufInit	1	123	123	0
tcpDoListen	1	122	122	0
getArgument	2	60	120	0
stringNCopy	12	9	118	0
enableAllInterruptsInContext	23	5	115	0
tmInit	1	110	110	0
tcpAlloc	6	17	102	0
netAlloc	4	24	96	0
vmAlloc	4	23	92	0
setClockValue	1	88	88	0
netcfgarpAddCacheETHIP	1	87	87	0
tcpTCBAllocate	2	40	80	0
stringCopy	12	6	74	0
tcpSynReceived	1	74	74	0
ethertap_handleMsg	2	36	73	0
tcpOpen	1	72	72	0
ioOpen	3	23	69	0
ioGetAddr	1	66	66	0
stringNCompare	1	65	65	0
shellMain	1	64	64	0
netarpUp	1	61	61	0
tcpEchoDeamon	1	61	61	0
netInit	1	57	57	0
nettap_setaddr	1	55	55	0
tmYield	5	11	55	0
netmodInit	1	54	54	0
tcpSync	1	54	54	0
netifSetAttr	4	13	52	0
ioInit	3	17	51	0
mmInit	1	48	48	0
tmSetStackPointer	23	2	46	0
tmSetReturnAddress	23	2	46	0
tmSetProgramCounter	23	2	46	0
tmSetStatusRegister	23	2	46	0
tmSetFramePointer	23	2	46	0
tmSetArgument0	23	2	46	0
netbufTrim	2	22	45	0
ethertap_init	2	22	45	0
netifInit	1	42	42	0

errLog	1	41	41	0
schedulerInit	1	41	41	0
roundUp	5	8	40	0
tcpServer	1	39	39	0
nettcpInit	1	38	38	0
mmInstallErrorHandlers	1	38	38	0
ioConsoleInit	1	38	38	0
netarpMain	1	37	37	0
display	2	18	36	0
tcpFragmentInsert	2	18	36	0
tcpRcvMaxSegSize	1	36	36	0
tcpNewSocket	2	17	34	0
lockInit	16	2	32	0
tmInstallErrorHandlers	1	30	30	0
mmUnmapPages	14	2	28	0
tmInstallExceptionCode	1	28	28	0
netudpMain	1	27	27	0
InitLinkBOARD	1	27	27	0
warning	1	25	25	0
devFPGA_init	1	25	25	0
devLoopback_init	1	25	25	0
netethInit	1	23	23	0
neticmpMain	1	23	23	0
hmInit	1	23	23	0
getCommand	1	22	22	0
tcpWindowInit	1	22	22	0
main	1	22	22	0
tcpSndMaxSegSize	1	21	21	0
tmUserTimerInit	1	20	20	0
netipInit	1	18	18	0
tcpInitialSequence	1	18	18	0
tmUserTimerGetTicks	1	18	18	0
ioSubscribe	1	17	17	0
netarpInit	1	16	16	0
neticmpInit	1	16	16	0
mmVmGetHeapAddress	2	8	16	0
dummyInterruptHandler	1	15	15	0
tmUserTimerThread	1	14	14	0
netdbgInit	1	14	14	0
getCurrentThreadName	1	14	14	0
tcpListenQueue	1	12	12	0
main	1	11	11	0
__start	1	10	10	0
lockInit	4	2	8	0
nettcpEchoTest	1	4	4	0
tmGetTickCount	1	4	4	0
tmUserTimerInitThreadId	1	3	3	0
__main	1	2	2	0
__main	1	2	2	0

Tabelle 5.3-1, Profiler-Ausgabe

6 Zusammenfassung

6.1 Schlussfolgerungen

Es stellte sich als richtigen Entscheid heraus, TCP zuerst auf MacOS zu portieren. Da jede Zeile Code neu geschrieben wurde, haben sich einige Fehler eingeschlichen, die auf Topsy schwierig zu finden gewesen wären. Dazu wusste man bei der Übernahme des Codes auf Topsy, dass die TCP-Implementation von der Funktionsweise her in Ordnung war. Dies schränkte die Möglichkeiten, wo sich Fehler befinden konnten, bedeutend ein.

Der Entscheid für die Portierung war ebenfalls grundlegend für die erfolgreiche Implementierung. Die vollständige Neuimplementation wäre sicher interessant gewesen, hätte jedoch in dieser Zeit kaum erfogreich ausgeführt werden können. Ein Subset von TCP hätte kaum Sinn gemacht, da dadurch die Kommunikation zu anderen Betriebssystemen mit TCP/IP Stack nicht möglich geworden wäre.

6.2 Ausblick

6.2.1 Verbesserungsvorschläge

6.2.1.1 Topsy-Integration

Die Integration der TCP-Implementation in Topsy könnte an verschiedenen Stellen verbessert werden.

- Ports
  Die Kommunikation zwischen dem Input- und Output-Thread geschieht zur Zeit über ein eigenes Ports-Modul. Dieses erlaubt das Versenden und Empfangen von Messages zwischen zwei Prozessen. Dies wird ähnlich auch von Topsy mit *tmMsgSend* und *tmMsgRcv* unterstützt. Um diese Funktionen verwenden zu können, müssten jedoch ein paar Anpassungen vorgenommen werden. Ports unterstützen z.B. auch eine Queuelänge und sind für das Senden blockierend, falls diese voll ist. Dies wird bei TCP auch zur Prozesssynchronisation verwendet.
- Speicherverwaltung
  TCP verwendet an verschiedenen Orten lineare Buffer (z.B. Sende- und Empfangsbuffer). Diese werden mit netbufAlloc angelegt im Wissen, dass bei diesen Grössen der NetBuf nur aus einem Buffer besteht und nicht verkettet ist. Grössere lineare Buffer, als sie von NetBuf unterstützt werden, würden für die Übertragung von grösseren Datenmengen die Performance deutlich erhöhen.
- Semaphoren
  Das Semaphoremodul unterstützt zur Zeit keine Prozesssynchronisation.

6.2.1.2 Performance

Es wurde kein Performance-Tuning vorgenommen. Vor allem im Bereich der Prozesskommunikation bzw. Prozesssynchronisation könnte sicher noch etwas herausgeholt werden.

6.2.2 Mögliche Erweiterungen

6.2.2.1 Socket-Layer

Der Socket-Layer unterstützt im Moment nur TCP. Er könnte jedoch erweitert werden für UDP und beispielsweise auch Raw IP.

6.2.2.2 Remote-Shell

Es sollte mit geringem Aufwand möglich sein, eine Remote Shell über TCP/IP für Topsy zu implementieren.

7 Anhang

7.1 Literaturverzeichnis

[AUF]	Aufgabenstellung
[GFR]	Topsy, A Teachable Operating System, G. Fankhauser
[K&R]	The C Programming Language, Kernigham/Ritchie
[SMC]	Code Complete, Buch zur strukturierten Software-Entwicklung, Steve McConnel Microsoft Press
[DC2]	Internetworking with TCP/IP, Volume II, Douglas E. Comer und David L. Stevens, Prentice Hall
[DWS]	A leightweight and high-performance TCP/IP Stack for Topsy, D. Schweikert

7.2 Glossar

ARP	Address Resolution Protocol; Internetprotokoll auf Layer 3
ICMP	Internet Control Message Protocol; Internetprotokoll auf Layer 4
IP	Internet Protocol; verbindungsloses Datentransportprotokoll auf Layer 3
OpenTransport	Name der Netztwerk-Unit im MacOS
Piggybacking	Verpacken von mehreren Informationen in ein TCP Paket. Beispiel: ACK zusammen mit Daten
TCB	Transmission Control Block; pro TCP Verbindung existiert ein TCB, in welchem alle verbindungsrelevanten Daten gespeichert werden
TCP	Transport Control Protocol; verbindungsorientiertes, Data-Stream Protokoll; Internetprotokoll auf Layer 4
UDP	User Datagram Protocol; verbindungsloses Internetprotokoll auf Layer 4
XINU	Name von Unix-ähnlichem Schulsystem. XINU ist die umgekehrte Schreibweise von UNIX

7.3 Abbildungsverzeichnis

7.4 Tabellenverzeichnis

7.5 Index