Der Retransmission Timeout von TCP

Philipp Lämmel
Betreuerin: PD Dr. Katinka Wolter

Inhaltsverzeichnis

Abstract

Diese Seminararbeit behandelt den Retransmission Timeout von TCP. Zuerst gebe ich einen Überblick über die Funktionsweise vom Transmission Control Protocol, sodass ich danach auf den Timer, samt seiner Berechnung und der Problematik, die hinter dem Algorithmus steckt, eingehen kann. Abschließend lege ich die mögliche Anwendbarkeit des Algorithmus in drahtlosen Netzwerken dar.

1 Einleitung

Das *Transmission Control Protocol* (TCP), welches im OSI-Modell in der Transportschicht liegt, ist für die zuverlässige Übertragung von Daten zuständig. Durch einen Three-Way Handshake wird die Verbindung hergestellt und die Übertragungsphase kann beginnen. Mit dem slow-start-Algorithmus wird die Fenstergröße auf ein Optimum eingestellt, so dass möglichst effizient verschickt werden kann. Ist nach einer bestimmten Zeit keine Bestätigung für das gesendete Paket angekommen, geht das Protokoll von einem Fehler aus und verschickt das Paket nochmal. Die Frage ist, wie lange der Timer bis zum erneuten Senden warten soll. Ist die Zeit zu groß, wartet das Protokoll unnötig lange auf das Ablaufen des Timers. Ist diese allerdings zu gering, wäre es möglich, dass das Paket erneut vergebens gesendet wird, da die Bestätigung zu lange brauchte. Sind alle Daten übertragen wird die Verbindung beidseitig geschlossen und die Übertragung ist mit dem „Last-Ack“ beendet.

2 Funktionsweise TCP

Das Transmission Control Protocol wurde ab 1973 von Robert E. Kahn und Vinton G. Cerf über mehrere Jahre hinweg entwickelt. 1981 gab es die erste Standardisierung [3]. TCP ist ein end-to-end reliable Protocol, das heißt, die Interaktion beruht vollends auf den beiden Endpunkten - sprich Sender und Empfänger und die Übertragung über mehrere Hops hinweg ist zuverlässig, da der Empfang bestätigt wird.
Der Austausch geschieht in Form von Segmenten, deren Gesamtgröße intern bestimmt wird. Theoretisch könnten die Segmente eine Größe von bis zu 65535 Byte haben, da sie dann noch in den IP-payload passen [8]. De facto ist die Größe der Segmente aber durch die *Maximum Transmission Unit* (MTU), welche mehrere tausend Byte groß ist, bestimmt [8]. Jedes Segment hat einen 20 Byte großen Header, der die wichtigsten Informationen, wie zum Beispiel die Sequenznummer, Quell- und Zielport, sowie die Flags enthält, wie man aus Abbildung 1 entnehmen kann.

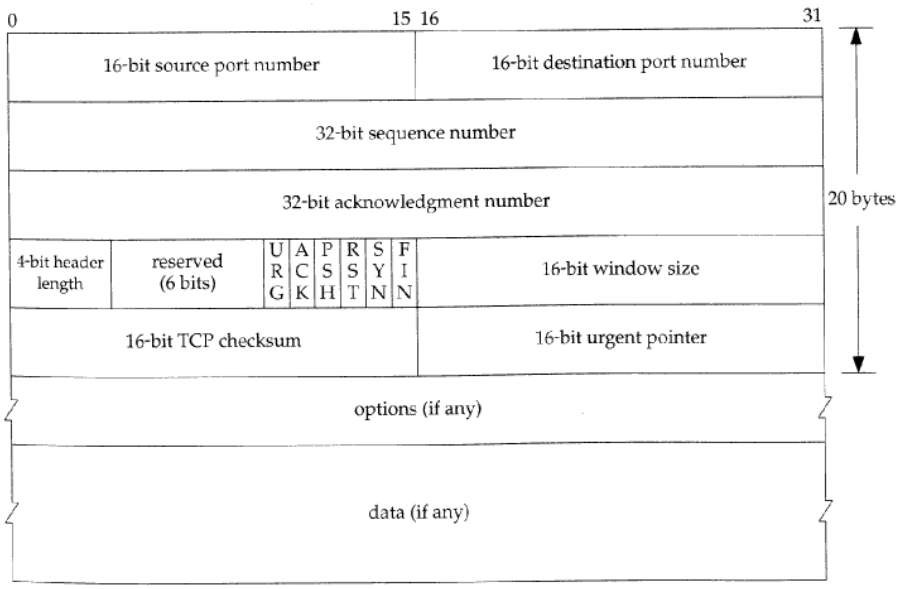

Abbildung 1: TCP header. [8]

2.1 Verbindungsablauf

Eine TCP-Verbindung ist eindeutig gekennzeichnet durch eine Quell-IP und -port sowie eine Ziel-IP und -port. Eine Verbindung wird mittels des Three-Way-Handshakes hergestellt, wobei der Server im Listen-Modus sein muss. Wäre der Server nicht im Listen-Modus, hätte er zusätzlich die Möglichkeit ein Paket mit gesetztem RST-Flag (RST für RESET) zurückzuschicken. Damit verdeutlicht der Server, dass keine Verbindung erwünscht ist.

Der Three-Way-Handshake funktioniert wie folgt:

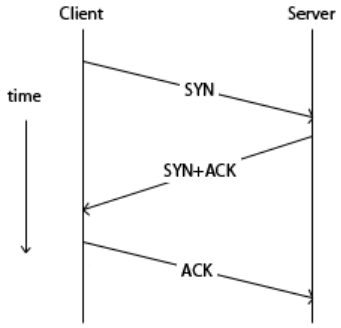

Abbildung 2: Three-Way-Handshake

Der Client schickt einen Connection Request mit gesetztem SYN-Flag (SYN für Synchronisation) und der Initial Sequence Number (ISN) an den Server. Das SYN-Flag äußert dem Server den Wunsch des Clients, eine Verbindung herzustellen. Darauf schickt der Server ein Paket mit gesetztem SYN- und ACK-Flag (ACK für Acknowledgement) als auch seiner eigenen ISN und bestätigt damit den Connection Request des Clients. Der Client sendet erneut ein Paket mit gesetztem ACK-Flag, bestätigt somit die ISN des Servers und die Verbindung ist komplett hergestellt. Es beginnt die vollduplexe Datenübertragungsphase, in der sowohl Client als auch Sever Daten schicken und empfangen können [8, 9].

Nachdem die Verbindung erfolgreich hergestellt wurde, wird nun die maximale Übertragungsmenge und dadurch die optimale Fenstergröße (*window size*) bestimmt. Diese gibt an, wie viele Daten auf einmal geschickt werden können bis der Client auf eine Bestätigung wartet. Das hintereinander Verschicken mehrerer Daten mit einem Mal, hat den Vorteil, dass man damit die Verzögerung für die folgenden Pakete reduziert und somit die Effizienz steigert. Initialisiert wird dieser Wert durch die maximale Segmentgröße (*Maximum Segment Size*). Wenn diese bestätigt wird ohne erneut gesendet zu werden, wächst die Fenstergröße exponentiell und zwar solange, bis entweder ein Timeout auftritt oder die maximale Fenstergröße des Servers erreicht ist. Bei einem Timeout kehrt das Protokoll zu dem letzten Wert, bei dem

eine Übertragung erfolgreich war, zurück. Dieses Verfahren wird slow-start genannt [8]. Der Congestion-Avoidance Algorithmus steht im engen Zusammenhang zum slow-start. Es gibt zusätzlich eine maximale Schwelle für die Fenstergröße, welche mit 64K initialsiert wird [8]. Ist die Fenstergröße bei dieser angekommen, wächst die Größe nur noch linear. Das lineare Wachstum geschieht solange, bis die maximale Fenstergröße des Servers erreicht ist. Tritt ein Timeout auf, wird der aktuelle Wert der Schwelle halbiert, die Fenstergröße wieder auf die Maximum Segment Size zurückgesetzt und die Größe pendelt sich erneut mittels slow-start auf ein Optimum ein. Dadurch verringert sich die Anzahl der Timeouts und die Netzwerküberlastung (*Congestion*) wird reduziert.

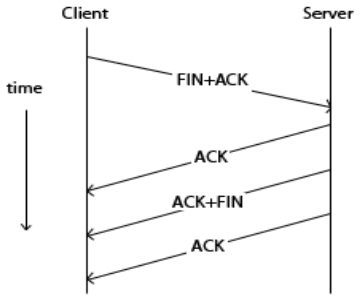

Abbildung 3: Verbindungsabbau

Sind alle Daten übertragen, sendet der Client ein Paket mit gesetztem ACK- und FIN-Flag (FIN für Finish). Das ACK-Flag bestätigt den Empfang aller Datenpakete. Daraufhin sendet der Server wieder ein Paket mit gesetztem ACK-Flag zurück und bestätigt somit den gewünschten Verbindungsabbau. Des Weiteren schickt der Server erneut ein Paket mit gesetztem FIN- sowie ACK-Flag und bereitet den Verbindungsabbau vor, in dem er die Fenstergröße auf 0 setzt. Die Verbindung wird mit einem Paket mit gesetztem ACK-Flag, welches auch als „Last-Ack“ bezeichnet wird [8, 9], beendet.

Es stellt sich die Frage, was mit Segmenten passiert, die bei der Übertragung verloren gehen und wie das Protokoll entscheidet, wann ein Segment neu gesendet werden muss.

3 Der Retransmission-Timeout

Der Retransmission-Timeout ist ein Timer, der dazu dient, zuverlässig bestätigen zu können, ob ein Segment erfolgreich übertragen wurde. Nach Ablauf des Timers, wird das Segment, auf dessen Bestätigung gewartet wird, erneut gesendet. Der Timer wird dynamisch berechnet.

3.1 Berechnung

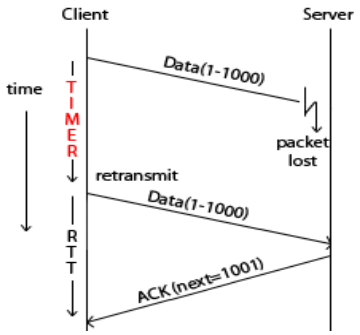

Abbildung 4: RTT und RTO.

Die Berechnung des Timers wurde in dem RFC 2988 festgelegt [5]. Die *round-trip time* (RTT) gibt die Zeit an, die vom Senden des Segmentes bis zum Empfang der Bestätigung vergeht. Die *smoothed round-trip time* ist eine Glättung der RTT und sorgt dafür, dass Extrema gemildert werden. Die *round-trip time variation* gibt an, wie stark die Berechnung streut.
In den folgenden Gleichungen stehen RTO für den retransmission timeout, SRTT für die smoothed round-trip time, σ für die round-trip time variation und G für die Messgenauigkeit des Timers (*clock granularity*). Bei der Initialisierung sollte der Wert des Timers, je nach Implementierung, zwischen $2,5 - 3$ *sec* betragen [5].

Nach der ersten Messung der round-trip time R^1, werden die Variablen wie folgt verändert:

$$SRTT = R^1 \quad (1)$$

$$\sigma = \frac{R^1}{2} \quad (2)$$

$$RTO = SRTT + max(G, 4 \cdot \sigma) \quad (3)$$

Bei den weiteren ausgewerteten Messungen R^i für die round-trip time werden die Variablen auf die folgenden Werte gesetzt:

$$\sigma = (1 - \beta) \cdot \sigma + \beta \cdot |SRTT - R^i| \quad (4)$$

$$SRTT = (1 - \alpha) \cdot SRTT + \alpha \cdot R^i \quad (5)$$

Da in (4) SRTT als Variable vorkommt, ist es unabdingbar, dass die Gleichungen in dieser Reihenfolge ausgeführt werden. Des Weiteren sollten die Werte für $\alpha = \frac{1}{8}$ und $\beta = \frac{1}{4}$ betragen [2], da dadurch die Effizienz maximiert wird. Letztendlich wird dann der RTO wie in (3) geändert.

3.1.1 Karn-Algorithmus

In diesem Algorithmus, der nach Phil Karn benannt ist, werden jene RTT R^i ignoriert, bei denen das Segment wiederholt gesendet wurde [4]. Dadurch wird dem Problem entgangen, dass man bei einem erneut gesendeten Segment nicht genau bestimmen kann, ob das erste, zweite oder das i-te bestätigt wurde und daher die Übertragungszeit nicht bekannt ist. Ein weiterer wichtiger Punkt ist, dass man RTO back-off wie folgt verwendet [4]:
Vor der erneuten Sendung eines unbestätigten Segmentes wird bei jedem auftretenden Timeout der RTO um einen Faktor a erhöht. Dies geschieht solange, bis das Segment bestätigt wird und der RTO pendelt sich wieder auf den vom SRTT-abhängigen Wert ein.

4 Probleme des TCP-Timers

Ein Timer ist immer dann notwendig, wenn man ein zuverlässiges verteiltes System aufbauen möchte [10]. Jedoch hat jeder Timer auch seine Probleme, die man in allgemeine und spezielle, auf den RTO bezogene, unterscheiden muss.

Zu den allgemeinen Problemen gehört, dass der Client zu Beginn der Übertragung keine Informationen über die Beschaffenheit des Netzwerkes hat. Dies bedeutet, dass er zum Beispiel nicht weiß, ob aktuell eine Überlast vorhanden ist. Daraus folgt das spezielle Problem eines angemessenen Initialwertes für SRTT und folglich auch für den RTO. Ist dieser zu lang, wird unnötig auf den Timer gewartet und die Effizienz leidet enorm [10]. Wenn der Initialwert aber zu kurz gewählt wurde, führt dies zu überflüssigen Neusendungen der Segmente. Das kann dann zu einer erhöhten Congestion führen [10].

Ein weiteres allgemeines Problem ist die Ursache für das Ablaufen des Timers. Diese ist nicht genau bestimmbar, da nur die Möglichkeit besteht, das Fehlschlagen festzustellen. Dies ist auch das zentrale Problem des TCP-RTO. Das Protokoll hat nicht die Möglichkeit zwischen verschiedenen Ursachen zu unterscheiden. Es geht im Allgemeinen davon aus, dass der Grund für das Ablaufen des Timers eine zu hohe Congestion ist, da Datenverluste in drahtgebundenen Netzwerken selten sind [1, 6–8]. Die Algorithmen aus 2.1 und 3.1 wurden deshalb auf Congestion angepasst.

Ein anderer Aspekt ist, dass der Algorithmus ausschließlich auf RTT Messungen beruht, welche dann SRTT, σ und RTO wie in (3) bis (5) bestimmen. Durch diese Gleichungen spiegelt der RTO aber nur eine durchschnittliche, jedoch keine momentane Verzögerung des Netzwerkes wider [4, 6]. Daher kann der RTO nicht effektiv auf die aktuelle Situation reagieren.

Da, wie bereits erwähnt, der Timer Congestion als Hauptquelle für Timeouts ansieht, vernachlässigt er in drahtlosen Netzwerken auch die network contention, welche auftritt, wenn mehrere Daten zeitgleich per Funk transportiert werden sollen. Dies hat Auswirkungen auf die Anzahl der erneuten Übertragungen, wie man Abbildung 5 entnehmen kann. Steigt die Anzahl an Daten, die gleichzeitig verarbeitet werden sollen, steigt auch die Anzahl an wiederholten Sendungen extrem an [6].

Doch auch in drahtlosen Netzwerken führt das Problem, dass Überlast als

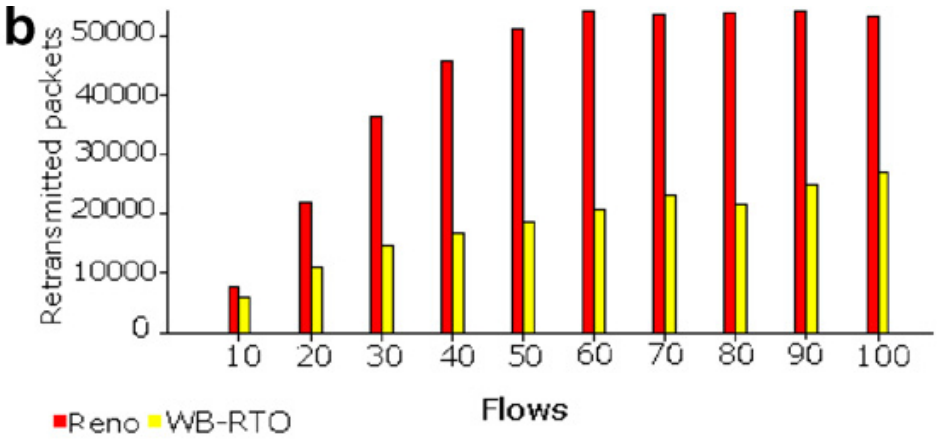

Abbildung 5: Retransmitted packets. [6]

Hauptquelle angesehen wird, zu Performanceeinbußen. Wie man in Abbildung 6 erkennen kann, kann ein Timeout auf Grund eines Blackouts auftreten, was in drahtlosen Netzwerken als Ausfall eines Knoten (node) interpretiert werden kann und nicht unüblich ist. Jedoch fehlinterpretiert das Protokoll den Timeout, geht von starker Congestion aus und erhöht dadurch den Timer. Es wird unnötig lange auf das Beenden des Timers gewartet:

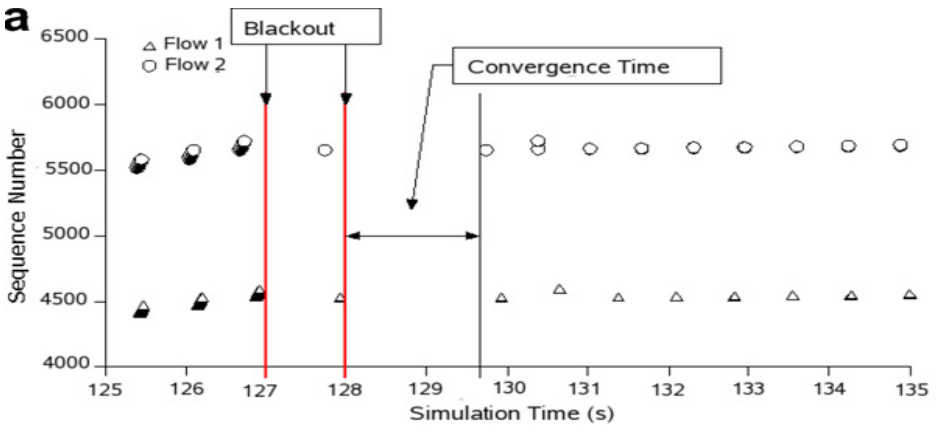

Abbildung 6: RTO nach Blackout. [6]

Da in drahtlosen Netzwerken viele Daten auf Grund von Datenverlust erneut gesendet werden müssen, stellt das Abarbeiten dieser Pakete ein wei-

teres Problem dar. Die Algorithmen verhalten sich ineffektiv und liefern in drahtlosen Netzwerken eine schlechte Performance [1].

5 Bewertung und Anwendbarkeit

Trotz der Probleme ist der Algorithmus zur Bestimmung des Timers in drahtgebundenen Netzwerken zuverlässig und stabil, da er auf Bestätigungen der gesendeten Segmente wartet und dadurch eine zuverlässige Übertragung gesichert ist. Auf Congestion bedingte Netzwerkveränderungen wird durch effizientes Anpassen des RTO-Wertes eingegangen. Wie im Abschnitt 4 bereits aufgeführt, ist zu hohe Congestion der vornehmliche Grund für das Auslösen des Timers. Jedoch gilt das in drahtlosen Netzwerken nicht, da in diesen Datenverlust, zum Beispiel durch Kollision, der Hauptgrund ist [1,7]. Das führt zu einer nicht tragbaren Performance in drahtlosen Netzwerken [6, 8] und man muss daher davon ausgehen, dass die TCP-Algorithmen nicht das Optimum in diesen Netzwerken darstellen.

Jedoch existieren diverse Erweiterungen des Protokolls, welche dieses Problem beheben. Eines davon ist das von Xia Gao, Suhas N. Diggavi und S. Muthukrishnan kreierte Link Layer Protection protocol (LHP), welches auf TCP beruht und in [1] vorgestellt wird.
Durch eine Explicit Loss Notification (ELN) kann das Protokoll Datenverluste unterscheiden, welche entweder auf Grund von link failure oder congestion ausgelöst werden. Um die ELN zu realisieren wird der Link Layer Header geschützt [1]. Wenn dieses Headerpaket ankommt, sogar wenn das IP-Paket korrupt ist, weiß man, dass es einen Datenverlust auf Grund von link failure gab. Diese Information kann dann genutzt werden, um ein ACK an den Sender zurückzuschicken, sodass dieser weiß, dass keine Congestion vorherrscht und der Timer nicht erhöht werden muss. Weitere Eigenschaften des LHP sind, dass es per stop-and-wait überträgt [1]. Dadurch können die Pakete nicht in falscher Reihenfolge ankommen, da immer erst gewartet wird, bis das vorherige Paket bestätigt wurde.

In den folgenden Abbildungen stehen reno für das Standard-TCP und (n)lhp für das LHP. SACK und HACK sind weitere Erweiterungen des TCP, welche aber in dieser Arbeit nicht von Bedeutung sind. Goodput gibt die Anzahl an Paketen an, die ohne Duplikat beim Empfänger angekommen sind.

Wie man aus Abbildung 7 entnehmen kann, ist beim LHP auch bei steigender Channel Error Rate die Anzahl an Paketen auf einem hohen Niveau, wohingegen beim TCP die Anzahl an Paketen, die kein Duplikat besitzen, stark verringert wird. Das liegt an dem in Abschnitt 4 genannten Problem, dass TCP nicht zwischen den Ursachen für das Ablaufen des Timers unterscheiden kann.

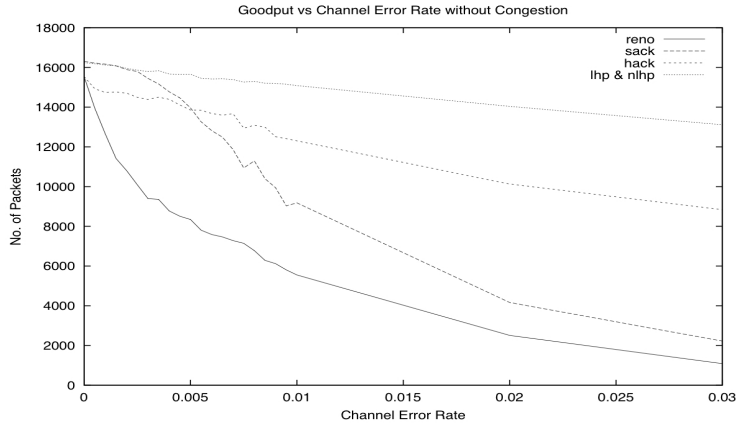

Abbildung 7: Goodput. [1]

In Abbildung 8 ist erkennbar, dass auch bei steigender Channel Error Rate und einer zusätzlichen fixen Congestion LHP nur wenige Duplikate versendet. Jedoch sieht es bei TCP ähnlich wie in Abbildung 7 aus. Der Grund dafür ist der selbe.

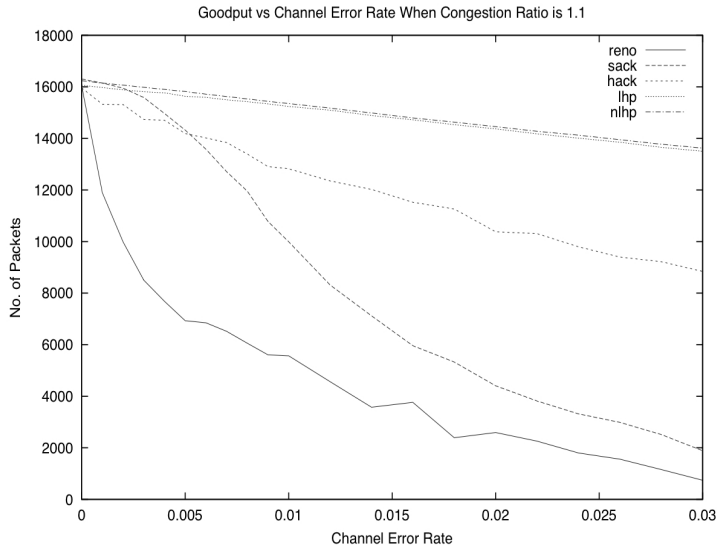

Abbildung 8: Goodput. [1]

Es ist festzuhalten, dass es Erweiterungen des TCPs gibt, welche in drahtlosen Netzwerken besser geeignet sind als das herkömmliche Transmission Control Protocol. Ein Beispiel dafür ist das oben eingeführte Link Layer Protection Protocol.
Weitestgehend alle Erweiterungen haben das Protokoll um die Funktion, zwischen den Ursachen des Datenverlustes zu unterscheiden, ergänzt, da dies der Hauptgrund für die schlechte Performance in drahtlosen Netzwerken ist.

6 Literaturverzeichnis

[1] X. Gao, S. N. Diggavi, and S. Muthukrishnan. Lhp: An end-to-end reliable transport protocol over wireless data networks. In *ICC '03 Communications*, May 2003.

[2] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, Aug. 1988.

[3] R. E. Kahn and V. G. Cerf. Transmission control protocol - darpa internet program protocol speficiation. In *RFC 793*, September 1981.

[4] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of SIGCOMM '87*, Aug. 1987.

[5] V. Paxson and M. Allman. Computing tcp's retransmission timer. In *RFC 2988*, May 2000.

[6] I. Psaras and V. Tsaoussidis. Why tcp timers (still) don't work well. In *Computer Networks: The International Journal of Computer and Telecommunications Networking*, volume 51, June 2007.

[7] F. Stann and J. Heidemann. Rmst: Reliable data transport in sensor networks. In *Proceedings of the First IEEE Sensor Network Protocols and Applications*, May 2003.

[8] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[9] K. Washburn and J. Evans. *TCP/IP Aufbau und Betrieb eines TCP/IP-Netzes*. Addison-Wesley, 1994.

[10] L. Zhang. Why tcp timers don't work well. In *Proceedings of SIGCOMM '86*, Aug. 1986.