Netzwerk-Tuning

Der TCP-Stack von Linux bietet zahlreiche Eingriffsmöglichkeiten, die sich auch für das Performance-Tuning nutzen lassen. Allerdings ist das Know-how nötig, das dieser Beitrag vermittelt, um an den richtigen Schrauben zu drehen.

Mattias Wadenstein

Es gibt im Wesentlichen zwei Gründe, sich ans Netzwerk-Tuning zu wagen. Zum einen: Man möchte einzelne Transfers beschleunigen. Das gelingt etwa durch gleichzeitiges Senden und Empfangen. Die versteckten Kosten bezahlt man hier meist in Form vermehrter Plattenaktivität. Aus Sicht der Platte wäre nämlich reines Senden oder Empfangen, das jeweils nur einen kontinuierlichen Schreib- oder Lesestrom bedingt, die performancefreundlichere Variante. Muss die Disk beides gleichzeitig erledigen, sinkt ihre Leistung.

Der zweite Grund für Netzwerk-Tuning ist: Es sind viele Clients mit Daten zu versorgen. Hier spielt die Anzahl konkurrierender Verbindungen die Hauptrolle. Besonders betroffen sind wieder die Platten, die die Daten dafür liefern oder aufnehmen müssen, aber auch der Speicher und andere wertvolle Ressourcen werden umso stärker beansprucht, je mehr gleichzeitige Verbindungen es gibt.

Beide Motive treffen sich in einem Punkt: Je schneller die Clients sind, desto eher sind sie fertig und das hat eine positive Auswirkung auf die Performance, denn am Ende des Transfers geben sie die verwendeten Ressourcen wieder frei. So fügt sich zusammen, was auf den ersten Blick wie ein Widerspruch erscheinen mag. Schnelle Clients führen zu weniger konkurrierenden Verbindungen und schonen damit unterm Strich die Ressourcen.

Schnelle Clients verlangen einen schnellen TCP-Stack. Dessen Funktionsweise ist zwar gut bekannt, allerdings führen die Wechselwirkungen zwischen unterschiedlicher Hardware und verschiedenen Implementierungen in der Praxis doch eher zu komplexen Problemen, bei denen die Real-Life-Performance deutlich von theoretischen Erwartungswerten abweichen kann.

Theoretische Limits

Der Durchsatz kann niemals höher ausfallen, als das theoretische Limit. Deshalb ist der Ausgangspunkt eine Formel, die die Grenzen der TCP-Bandbreite absteckt: `bandwidth=min(window_size/RTT,MSS/(RTT*sqrt(loss)))`. Das bedeutet, die Bandbreite hängt von zwei Faktoren ab:

window_size/RTT: Die Fenstergröße ist mit der Latenz verbunden. Das TCP-Fenster ist der Puffer, der alle Pakete aufnimmt, deren Empfang die Gegenseite noch nicht bestätigt hat, die sich also quasi noch in der Leitung befinden. Per Default hatte Linux bis zum Kernel 2.6.17 eine Fenstergröße von 64 KByte, was – unter der Annahme einer Round-Trip Time von 150 ms, die mit »ping« leicht nachzumessen ist – eine maximale Internet-Bandbreite von 64/0.15 = 400 KByte/sec ergibt. Das ist sehr langsam für heutige Verhältnisse, etwa 20-mal langsamer als man das von einer normalen Fast-Ethernet-LAN-Verbindung erwarten würde.

MSS/(RTT*sqrt(loss)): Das ist der Paketverlust im Verhältnis zur Round-Trip Time und Paketgröße. Die maximale Segmentgröße (MSS) beträgt typischerweise 1460 Bytes (die 1500 Bytes eines normalen Ethernet-Frame minus Header) und unter der Annahme von 0,1 Prozent Paketverlust bei derselben Round-Trip Time von 150 ms ergibt das 1460/(0.15*sqrt(0.001)) ~= 300 KByte/sec.

Wer sich schon eingehender mit der Materie beschäftigt hat, weiß, dass diese Formel nur eine Näherung darstellt. Aber die exakte Berechnung ist hier gar nicht nötig, es geht um die Größenordnung, die mit keinem Tuning zu übertreffen ist. Doch es gibt ein paar Tricks – mehr dazu später.

Außerdem ist zu beachten, dass die Protokolle über dem TCP-Layer zuweilen mit gleitenden Fenstergrößen arbeiten, die manchmal nicht groß genug sind. Deshalb sollte man an diese Möglichkeit besonders dann denken, wenn die reine TCP-Performance gut zu sein scheint, der Durchsatz bei Transfers über den Applikationslayer aber einbricht.

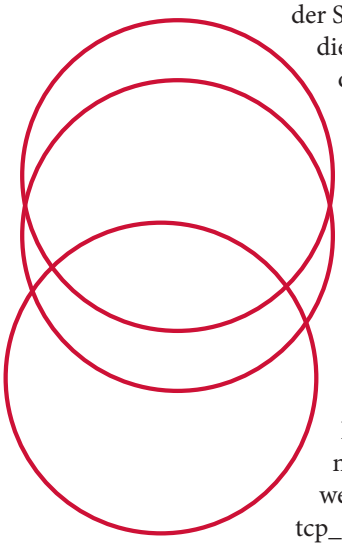
Praktisches Window-Tuning

Was beschränkt nun die Bandbreite am meisten? Wer keinen Kernel verwendet, der neu genug ist, um bereits das automatische Anpassen der Fenstergröße zu unterstützen, der muss definitiv die Fenstergröße manuell heraufsetzen. Seit der Kernelversion 2.6.17 praktiziert Linux ein Autotuning, aber alle früheren Versionen profitieren von beherztem Tuning.

Die Parameter, die dafür in Frage kommen, sind »net/ipv4/tcp_rmem« und »net/ipv4/tcp_wmem« – jeweils einer für das Senden und Empfangen. Sie sind in drei Felder aufgeteilt für Minimum, Default und Maximum. Für ältere Kernel empfehlen sich folgende Werte: ▶

```
net/core/rmem_max = 8738000
net/core/wmem_max = 6553600
net/ipv4/tcp_rmem = 8192 873800 8738000
net/ipv4/tcp_wmem = 4096 655360 6553600
```

Der Grund für den größeren Maximalwert für »tcp_rmem« ist, dass sich dessen effektive Größe um »window_size/2^tcp_adv_win_scale« vermindert. Per Default hat »tcp_adv_win_scale« den Wert 2, sodass eine Zugabe von 25 Prozent nötig ist, um auf denselben Effektivwert wie auf der Senderseite zu kommen. Diese Werte stellen die Fenstergröße auf 640 KByte ein, was für dieselbe Round-Trip Time wie oben einen viel eher akzeptablen maximalen Durchsatz von 4 MByte/s ergibt.



Für TCP-Fern-Transfers haben die Kernel seit 2.6.17 einen automatischen Tuning-Mechanismus. In diesem Fall kann man die Default-Fenstergröße klein lassen und braucht nur den dritten Parameter, um die maximale Fenstergröße anzupassen und ein größeres TCP-Fenster zu erreichen

Die mögliche Kehrseite eines vergrößerten Fensters ist, dass damit auch mehr Kernelspeicher für Puffer nötig ist. Normalerweise regelt das der Parameter » net/ipv4/tcp_mem« in »/etc/sysctl«, der den Speicher für verschiedene Puffer begrenzt. Setzt man ihn zu hoch, verwendet das Betriebssystem allen Speicher für TCP-Puffer und für Prozesse bleibt nichts mehr übrig.

Mehr TCP-Tuning

Wenn nicht die Fenstergröße, sondern der andere Faktor MSS/Loss die TCP-Performance begrenzt, ist ein Gegenmittel sehr viel schwerer zu finden. Zunächst gilt es festzustellen, wo genau der Paketverlust eintritt: im lokalen Interface, dem lokalen Switch, oder irgendwo im Internet.

Die erste Anlaufstelle ist das lokale Interface der Senderseite. Hier verrät »ifconfig«, ob das Interface irgendwelche TX-Pakete verworfen hat. In diesem Fall hilft es wahrscheinlich, die Länge der Sende-Warteschlange zu erhöhen. Auf der Empfängerseite kann theoretisch dasselbe passieren, allerdings ist die Wahrscheinlichkeit dafür erfahrungsgemäß geringer. Häufiger verwirft der Switch auf der Empfängerseite die Pakete. Die meisten Switches führen darüber allerdings Buch und diese Statistik lässt sich bei ihnen abfragen.

Bewahrheitet sich der Verdacht, kann man wenig tun. In diesem Fall ist davon auszugehen, dass den Switch über sein Uplink-Interface mehr Pakete auf einmal erreichen, als er wieder auszusenden in der Lage ist. Der Switch müsste die Pakete dann puffern, doch meistens ist der dafür verfügbare Speicher viel zu klein.

Eine Möglichkeit ist unter Umständen, die MSS zu erhöhen, typischerweise, indem man zu Jumbo-Frames wechselt. Das hilft zwar nur manchmal, ist aber einen Versuch wert, wenn man Sender- und Empfängerseite kontrolliert. Dabei ist immer daran zu denken, dass dann alle Hosts des betroffenen Subnetzes umzustellen sind, denn Linux beherrscht keine Path MTU Discovery für LAN-Traffic.

Einzelne Verbindungen zu beschleunigen, ist nur ein Teil des Netzwerk-Tunings. So benötigt ein viel beschäftigter Webserver sicherlich eher Hilfe, um viele gleichzeitige Verbindungen zu bewältigen. Dieses Problem unterteilt sich weiter in zwei Teilprobleme: Eine hohe Anzahl konkurrierender Verbindungen und eine hohe Verbindungsrate.

Hohe Verbindungszahlen

Eine große Anzahl konkurrierender Verbindungen zu beherrschen, ist in erster Linie ein Problem des Applikations-Tunings. Hier geht es beispielsweise darum Apaches »MaxClient«-Direktive so einzustellen, dass der Webserver aller Verbindungen Herr wird. Doch wenn der Server erst einmal langsamer wird, dann besteht die Gefahr, dass sich mehr und mehr Verbindungen anhäufen, Es entwickelt sich ein selbstverstärkender Effekt, solange bis der Server die Clients wieder schnell genug bedient und sie ihren Platz für neue Verbindungen räumen.

Zusätzlich zum Applikations-Tuning sind für hohe Verbindungszahlen auch ein paar Kernel-Parameter interessant. Der wichtigste ist »tcp_max_syn_backlog«, der festlegt, wie viele Verbindungen der Kernel annimmt, ohne dass sie die Applikation vorher akzeptiert hat. Der voreingestellte Wert ist 1024, was gut zu einem mittelstark beschäftigten Server passt. Rechnet man stattdessen mit Tausenden Requests in der Sekunde, ist es vorteilhaft, den Wert zu erhöhen.

Für Webserver mit ungefähr 1000 Requests in der Sekunde hat sich nach Erfahrung des Autors ein Wert von 8192 bewährt. Im Log sollte der Admin auf die Fehlermeldung »TCP: drop open request« achten, die besagt, dass die Anzahl der Slots für Verbindungen erschöpft ist, bevor sie

der Kernel vollständig aufbauen und an die Applikation im Userspace abgeben konnte.
Das Wichtigste für Server mit hohem Traffic ist, dafür zu sorgen, dass jeder Client schnellstmöglich bedient wird, ehe sich ein Stau aufbaut. Gleichzeitig sollte man aber auch die Anzahl konkurrierender Zugriffe nicht übermäßig beschränken, denn sonst können wenige Anwender mit Modems alle Slots blockieren. Die rechte Balance lässt sich typischerweise nur unter realer Last finden.

Network Stack Monitoring

Hat man mit hoher Netzwerklast zu kämpfen, ist es manchmal schwierig festzustellen, was unter der Haube passiert. Das zu wissen ist nicht immer nötig, zuweilen aber entscheidend, vor allem, wenn es den Anschein hat, dass sich daneben benehmende Clients alle Ressourcen verbrauchen. Dann ist es wichtig, an den richtigen Stellen nachzuschauen, denn es gibt auch eine Vielzahl von Fällen, die zwar auf den ersten Blick bedrohlich wirken, tatsächlich aber kaum zum Ressourcen-Verbrauch beitragen.

Das wichtigste Tool, um festzustellen, was gerade auf dem Netzwerk passiert, ist »netstat«, das alle momentanen Verbindungen und ihren Status zeigt. Zu verstehen, was die einzelnen Status im Zusammenhang des Netzwerk-Stack von Linux meinen, ist in den gewöhnlichen Fällen leicht, kann aber auch einigermaßen schwierig sein. Auf jeden Fall sind die in RFC 793 wohl definierten TCP-States ein guter Ausgangspunkt (**Abbildung 1**).
Die **Abbildung 1** illustriert die normalen TCP-States, zu denen sich noch zahlreiche Fehlerbedingungen gesellen. Die mögen in manchen Fällen auch entscheidend sein, wichtiger aber ist, eine Übersicht zu gewinnen, um die von »netstat« reportierten Status zuordnen zu können. Dabei ist zu beachten, dass »netstat« den Status »CLOSED« nicht kennt, im Diagramm steht er für den Zustand, in dem keine Verbindung besteht.
»Recv-Q« steht für die Anzahl Bytes in der Obhut des Kernels, die er noch nicht an die Userspace-Applikation weitergegeben hat, die mit diesem Socket verbunden ist. »Send-Q« steht

für die Anzahl von Bytes, die von der Gegenseite noch nicht bestätigt wurden. Das ACK (acknowledged) steht also noch aus. Manchmal sind diese ein Anzeichen hängender oder abgebrochener Verbindungen.

Die interessanteste Spalte ist mit »State« überschrieben; hier lassen sich die meisten Clients und Netzwerke enttarnen, die sich nicht richtig verhalten.

»**ESTABLISHED**« ist der normale Status einer aktiven Verbindung, in diesem Stadium werden Daten übertragen. Sind alle Daten ausgetauscht, wird die Verbindung geschlossen und verschwindet. Alles ist ok.

Eine übergroße Anzahl Verbindungen in diesem Status – besonders, wenn sie zu wenigen IP-Adressen gehören – kann auf fehlerhafte Clients hinweisen. Dann empfiehlt es sich die Anzahl der Verbindungen pro Client zu begrenzen.

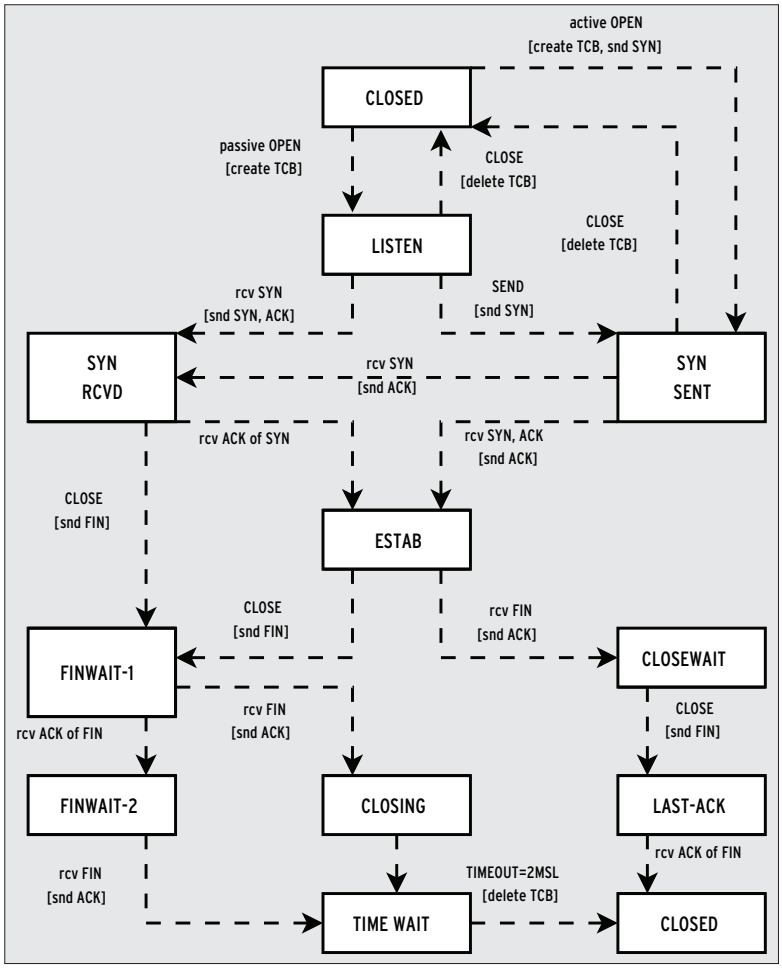


Abbildung 1: Der RFC 793 definiert, in welchem Status sich eine TCP-Verbindung befinden kann und welche Statusübergänge möglich sind.

»**SYN_RECV**« bedeutet, dass der entfernte Client eine Verbindung initiiert hat, indem er ein SYN-Paket sendete, der Handshake ist aber noch nicht abgeschlossen. Unter normalen Umständen mündet dieser Status schnell in »ESTABLISHED«, aber auf einem viel beschäftigten Server können sich eine nennswerte Anzahl Verbindungen in diesem Status befinden. Außerdem zeigen sich auch SYN-Flood-DoS-Attacken durch eine hohe Zahl halboffener Verbindungen und schließlich können auch fehlerhafte Firewalls ein Verweilen in diesem Status erzwingen. Selbst eine große Anzahl solcher Verbindungen ist für Linux allerdings in der Regel kein Problem, denn es reserviert keinen Speicher dafür. »**FIN_WAIT**« and »FIN_WAIT2« sind zwei Status, die durchlaufen werden, wenn der Server die Verbindung beendet. Dabei braucht »FIN_WAIT1« mehr Ressourcen, weil sich in diesem Zustand noch Daten auf der Leitung befinden können. »FIN_WAIT2« wartet lediglich auf das letzte FIN-Paket der Gegenseite, um danach die Verbindung zu beenden. Der Timeout für »FIN_WAIT2« lässt sich über »tcp_fin_timeout« einstellen. Weil aber jede Verbindung in »FIN_WAIT2« nur 1,5 KByte Memory beansprucht, lohnt sich das meistens nicht.

»**CLOSE_WAIT**« und »LAST_ACK« sind gleichartige Status, diesmal für den Fall, dass der Client (und nicht der Server) sich für den Abbau der Verbindung entscheidet. Für alle diese Status gilt, dass ihre Anzahl in einem gesunden Verhältnis zur Anzahl eingehender Verbindungsanfragen stehen soll. Eine übermäßig hohe Anzahl deutet auf eine defekte oder fehlkonfigurierte Firewall – oder einen Angriff.

»**TIME_WAIT**« ist ein Status, in dem die Verbindung eine Weile wartet. Das soll verhindern, das herumstreunende Pakete, die sich noch in der Leitung befinden, unbeabsichtigterweise einer neuen TCP-Verbindung zugerechnet werden. Dieser Status braucht sehr wenig Ressourcen und verursacht sehr selten Probleme. Das genaue Verhalten lässt sich über diverse Parameter steuern, was aber kaum je nötig ist.

In der »dmesg«-Ausgabe eines modernen Kernels finden sich zahlreiche Warnmeldungen. Manche davon sind wichtig, aber hier sind die, die jeder Admin sicher ignorieren kann:

- »TCP: Treason uncloaked!«
- »MD5 Hash NOT expected but found«
- Most IPv6 messages (selbst wenn man IPv6 benutzt). *(jcb)*