Performance Evaluation of Quick-Start TCP with a Linux Kernel Implementation

Michael Scharf and Haiko Strotbek

Institute of Communication Networks and Computer Engineering (IKR)
University of Stuttgart, Germany
michael.scharf@ikr.uni-stuttgart.de, haiko@strotbek.com

Abstract. Quick-Start is an experimental extension of the Transmission Control Protocol (TCP) that uses explicit router feedback to speed up best effort data transfers. With Quick-Start, TCP endpoints can request permission from the routers along the path to send at a higher rate than allowed by the default TCP congestion control, which avoids the time-consuming Slow-Start. However, since Quick-Start TCP requires modifications in the protocol stacks of end-systems and routers, realization complexity is a major concern. This paper studies Quick-Start with a new implementation in the Linux protocol stack. We first show that Quick-Start support can be added to a real stack with rather limited effort, without causing much processing overhead. Second, we perform measurements with Web applications and study the impact of important parameters. These experiments with real applications demonstrate that Quick-Start can significantly speed up data transfers, and they confirm the outcome of previous simulation efforts. Our results suggest that Quick-Start is a lightweight mechanism that could be very beneficial for broadband interactive applications in the future Internet.

1 Introduction

Most Internet applications use the Transmission Control Protocol (TCP) for reliable, best effort transport. Since TCP is a pure end-to-end protocol, the connection endpoints must continuously probe the available bandwidth on the path in order to adapt their sending rate. The TCP congestion control is challenged by new broadband technologies with large bandwidth-delay products. In such cases, TCP has difficulties in determining an appropriate data rate, in particular after connection setups, or after long idle periods. Several "clean slate" Internet research efforts address this issue by using explicit feedback from routers.

Quick-Start [1] is an experimental TCP extension that applies explicit router feedback to avoid the time-consuming Slow-Start [2], which is often considered to be a problem in high-speed networks. With Quick-Start, hosts can ask for a high initial sending rate, e. g., during the three-way handshake, and the routers along the path can approve, modify, or discard this request. Several simulation studies [3, 4] show that these mechanisms can significantly reduce the completion time of high-speed data transfers over paths with a large bandwidth-delay product, such as over broadband wide area networks, satellite or cellular links.

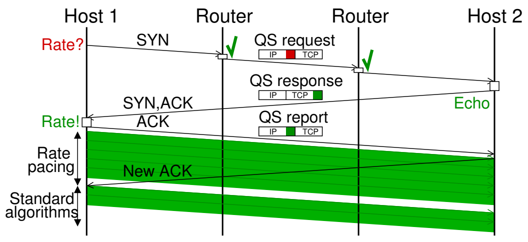


Fig. 1. Illustration of a Quick-Start (QS) request during the three-way handshake

The new Quick-Start TCP extension requires modifications of the protocol stacks, both in end-systems and in routers. Thus, similar to other related schemes for router-assisted congestion control, implementation and deployment complexity is a major concern. In order to answer the question whether Quick-Start could be useful in the future Internet, implementations and experiments with real applications are required [1].

This paper contributes to this discussion in two ways: First, we present a complete implementation of Quick-Start in the Linux kernel. To the best of our knowledge, it is the first operational Quick-Start implementation in a real-world protocol stack. We also report some of the lessons learned during our development work, extending an initial presentation in [5]. Second, we use our implementation to perform experiments with different applications and network scenarios. The results illustrate the usefulness of Quick-Start in real setups.

The rest of this paper is structured as follows: Section 2 introduces the Quick-Start extension and surveys related work. In Section 3, we present details of our implementation. Section 4 reviews our evaluation methodology. The results of our measurements are discussed in Section 5. Section 6 concludes the paper.

2 Quick-Start TCP Extension

2.1 Overview

The Quick-Start mechanism enables TCP connection endpoints to determine an allowed sending rate in cooperation with the routers on the path, in particular at the beginning of a data transfer. Fig. 1 illustrates a Quick-Start request during the connection establishment: In order to indicate its desired sending rate, the originator adds a *Quick-Start request* option to the Internet Protocol (IP) header. This target rate is encoded in 15 coarse-grained steps ranging from 80 kbit/s to 1.31 Gbit/s. The routers along the path can approve, modify, or disallow this rate request. If the request arrives at the destination, the granted rate is echoed back as a piggybacked TCP option (*Quick-Start response*). The originator then determines whether all routers along the path have approved the request. If not, the default congestion control (Slow-Start) is used to ensure backward compatibility. If the Quick-Start request is successful, the originator can immediately increase its congestion window and start to send with the approved rate. After one round-trip time (RTT) the Quick-Start phase is completed and the default TCP congestion control mechanisms are used for subsequent data transfers.

Quick-Start does not guarantee any data rate, i. e., it is a lightweight speedup mechanism for elastic best effort traffic only. Simulations [3] and analytical models [4] reveal that the transfer times of moderate-sized files can be improved by several hundred percent. In addition to avoiding initial Slow-Starts, the Quick-Start mechanism could also be quite useful in the middle of data transfers, e. g., after longer idle periods, or in combination with link layer mobility triggers.

However, Quick-Start TCP, as any router-assisted congestion control approach, comes at some cost: It requires support in *all* routers along a path, which causes processing overhead and which makes a short-term deployment in the Internet unrealistic. There are also interworking issues, e. g., with IP tunnels and firewalls. This is why [1] recommends that the use of Quick-Start should initially be limited to controlled environments such as intranets.

2.2 Related Work

There are numerous proposals that address the performance limitation of TCP's Slow-Start. A comprehensive survey can be found in [1]. Four principal approaches can be distinguished: (1) Schemes that apply bandwidth estimation techniques in order to determine the available bandwidth, (2) additional function blocks that share path capacity information, such as a "congestion manager", (3) explicit feedback from network elements along a path, and (4) proposals to use an arbitrarily high sending rate at the beginning of data transfers [6]. Quick-Start belongs to the third category. It does not suffer from the inaccuracy of end-to-end bandwidth estimation techniques. Unlike the second approach, Quick-Start can also handle completely new paths. And, different to the fourth category, there is no danger of severe congestion due to over-aggressiveness.

Quick-Start is to be used in transient phases only, when precise information about the path is missing. Other research approaches suggest replacing the TCP congestion control by a fine-grained, per-packet feedback from routers. Two examples are the eXplicit Control Protocol (XCP) and the Rate Control Protocol (RCP). XCP has been implemented in FreeBSD [7] and Linux [8]. In the latter case, the XCP protocol is encoded as a TCP option, and the router functions are realized as a loadable module for the Linux traffic control. A Linux implementation of RCP has recently been reported in [9]. Here, a shim layer has been added in the endpoints between IP and TCP, and the router support is provided by a Linux netfilter plugin. These implementation efforts demonstrate the need for practical experiments with such new congestion control schemes.

3 Quick-Start Implementation in the Linux Kernel

3.1 Required TCP/IP Protocol Stack Extensions

Quick-Start requires modifications in the stacks of end-systems as well as additional processing in every node that forwards IP packets towards potential

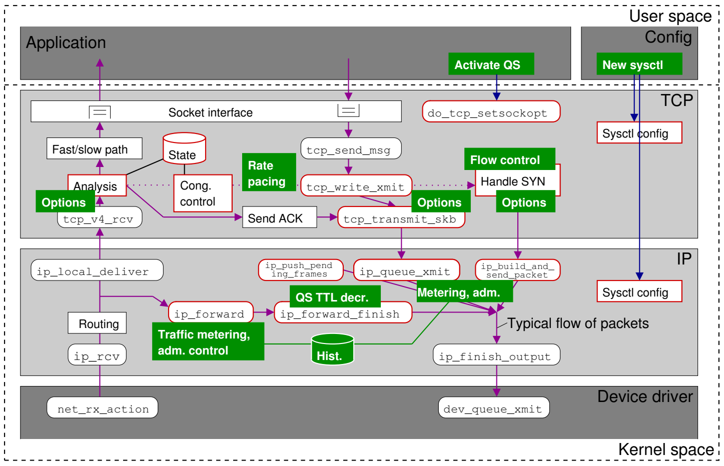


Fig. 2. Linux kernel code modifications by our Quick-Start implementation

bottleneck links. For instance, the originator must build and send the IP option, validate the response, and then use the allowed data rate when sending new data. This requires an additional rate pacing mechanism because TCP is a window-based protocol without rate control. The receiver must echo back the requests as a TCP option. Furthermore, the flow control may have to be changed [10].

Quick-Start enabled routers require three new functions: First, routers have to determine the capacity of the outgoing links and keep track of their utilization, unlike today's routers, which are not necessarily aware of link capacities. This information can be either obtained by configuration, by cross-layer information exchange with the sub-IP layer, or by bandwidth estimation techniques. Second, routers must process the new IP options, i. e., they must read and write some of the entries (e. g., the data rate). And third, routers must perform an admission control that decides whether to accept a Quick-Start request, and which data rate to grant. This typically requires a short history about recently approved requests. All three functions can be realized without keeping any per-flow state.

3.2 Implementation Details

The Linux kernel has a highly optimized TCP/IP networking stack. This stack is thus an excellent basis for studying the implementation effort caused by Quick-Start. We extended the Linux TCP/IP stack so that it completely implements Quick-Start for IP version 4 according to [1], using kernel version 2.6.20 as basis. The implemented features include both the end-system and router functions mentioned in the previous section. Fig. 2 gives a simplified overview of the structure of the IP and TCP layer in the Linux kernel, highlighting a couple of important functions that are involved in packet processing (details are given e. g. in [11]). The blocks in this figure illustrate our main modifications of the kernel code. Even though the Quick-Start mechanism is rather simple compared of the

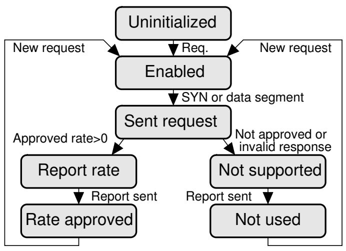


Fig. 3. Quick-Start state engine

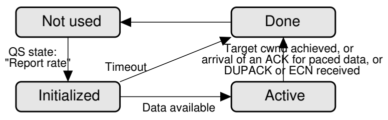

Not used
Done
QS state: "Report rate"
Timeout
Target cwnd achieved, or arrival of an ACK for paced data, or DUPACK or ECN received
Initialized
Data available
Active

Fig. 4. Rate pacing state engine

complexity of TCP as a whole, changes have been required in almost every part of the TCP implementation, and also in several IP functions. In the following, we briefly explain some aspects where implementing the specification [1] has not been straightforward. More details are documented in [12].

A Quick-Start sender must keep additional state concerning a request and the rate pacing. Our Quick-Start state engine is sketched in Fig. 3. Some complexity is caused by the requirement that requests must only be sent in IP packets carrying a TCP segment that gets acknowledged, i. e., with a SYN flag or with payload data. Further states are needed for the rate pacing (see Fig. 4), because the rate pacing only starts when new data is available, and because there are several abort conditions. We have realized the rate pacing by an additional timer. Each time the timer expires, the congestion window `snd_cwnd` is increased by a given amount of segments, up to the Quick-Start congestion window `cwnd_qs`. As Linux offers a high timer granularity of 1 ms when running with $\texttt{HZ} = 1000\,1/\text{s}$, there is some degree of freedom whether to use few or many timers. As shown in Fig. 5, one can distinguish three different cases: In case 1, the increase of `cwnd_qs` is small and the timers must be distributed evenly over `rtt_ticks` possible timers during one RTT. However, if `cwnd_qs` is large (case 2), the maximum number of timers `rtt_ticks` can be reached, and carryover segments might be needed if `cwnd_qs` is not a multiple of `rtt_ticks`. The overhead of many timers can be reduced by enforcing a minimum window increase per timer (case 3), i. e., a minimum chunk size `M`. The number of timers then follows as

$$\texttt{timer} = \min\left(\texttt{cwnd\_qs}/\texttt{M}, \texttt{rtt\_ticks}\right)$$

The configuration parameter `M` allows to trade off timer processing overhead vs. traffic burstiness. By default, we use a minimum chunk size of 3 segments.

The Quick-Start router functions are realized by a new method in the forwarding path of IP packets. This method meters the traffic during intervals of duration D (1 s, by default), and it processes the Quick-Start requests. We implemented the “target algorithm” admission control [1, 3]. The recently approved requests are stored in a ring buffer (cf. [4]). The default size is two slots, i. e., the granted bandwidth is remembered for $H = 2\,D = 2\,\text{s}$. An alternative solution would have been to use the Linux netfilter hooks. However, a precise traffic metering requires some additional processing of every packet anyway.

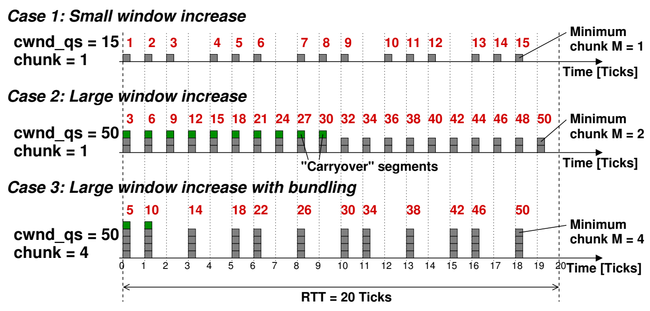


Fig. 5. Incrementing the congestion window with timers to realize rate pacing

3.3 Implementation Complexity and Lessons Learned

The experiments in the next sections show that our Quick-Start implementation is fully operational. The total implementation effort has been rather limited: The patch adds or modifies less than 2000 lines of kernel code ($< 5\,\%$ of the TCP and IPv4 code). It requires about 20 additional integer variables per TCP connection in the `tcp_sock` structure and some new variables per device in the IP layer. It is configured by some additional `sysctl` variables and `ioctl` calls.

The addition of Quick-Start results in some ugly code. Since Quick-Start violates the layer separation between IP and TCP, interface extensions are required between the layers. Options have to be processed in a couple of different functions so that many parts of the TCP/IP stack have to be modified. A tricky problem is that the TCP maximum segment size must be reduced for IP packets carrying a Quick-Start IP option in order to avoid packet fragmentation.

A remaining challenge is to automatically determine the capacity of outgoing links. There are certain possibilities to obtain such information from the corresponding Linux device drivers, but there is no "one-fits-all" interface from the network layer to the drivers, even for standard Ethernet network cards. Our current solution is to manually configure the link capacity for each interface.

These changes result in performance costs. The overhead introduced by the Quick-Start functions is rather small in typical usage scenarios. In the TCP layer, a fine-grained rate pacing requires additional timers. Yet, this does not have a significant impact on the system load unless there are many parallel connections.

The overhead in the IP layer is also small, as long as only few packets carry Quick-Start options. We performed a kernel profiling analysis to determine the load share of all method calls that handle the packet forwarding and process IP options. Fig. 6 presents the results obtained from the "oprofile" tool, using both an unmodified and a patched kernel. A router (P4 2.8 Ghz PC with Gbit/s Ethernet interfaces) was exposed to a small packet rate of 100 packets/s and a high load of 300, 000 packets/s. Both workloads use small 50 byte packets.

Without any Quick-Start options being present, there is a measurable overhead, but it is rather small. The additional effort is mainly caused by the traffic metering that counts every IP packet. It could be avoided if the link utilization

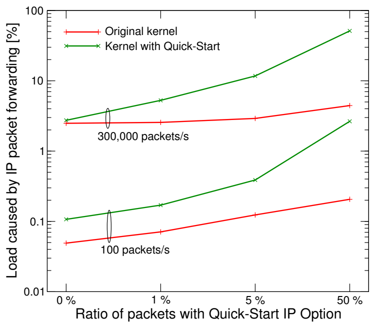


Fig. 6. Router processing overhead

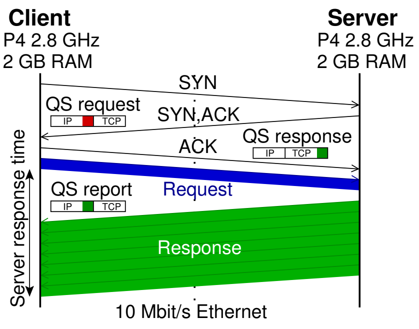


Fig. 7. Client-server communication

was already available, e. g., from network monitoring components. If 1 % of the packets include a Quick-Start option, which could be a realistic scenario, the option processing can increase the load by few percent. Still, this effect neither significantly decreases the maximum throughput (about 330, 000 packets/s), nor increases the packet delay in the router (ca. 50 μs in this scenario). Only if the share of Quick-Start packets is very high, the Quick-Start processing causes a considerable additional system load, which can mainly be attributed to the recalculation of the random Quick-Start nonces (see [1]). But such a traffic characteristic is unrealistic and should never occur when Quick-Start is used as foreseen.

4 Measurement Methodology

4.1 Testbed Setup

In order to test our Quick-Start implementation and to study the benefit compared to standard TCP, we used a network setup with a client and a server. As illustrated in Fig. 7, Quick-Start is activated for data transfers from the server to the client, with the request being sent in the SYN/ACK segment. Both computers were interconnected by a direct Ethernet link that is forced to 10 Mbit/s line speed. This is a realistic value for long-distance connections over today's access and wide area networks. Client and server are Ubuntu 7.04 installations running the modified Linux kernel. The Linux "NetEm" network emulation was used to delay packets for a constant duration, which models the latency of wide area networks. As recommended in [10], the socket buffers were increased to a larger value (8 Mbyte) in order to avoid any limitation by the TCP flow control.

4.2 Quick-Start Enabled Applications

A Quick-Start request can be activated by two mechanisms: First, we have implemented some kernel-internal heuristics that can automatically issue a Quick-Start request. For instance, a kernel can be configured to send a request in every TCP connection setup. The request can also automatically be repeated after

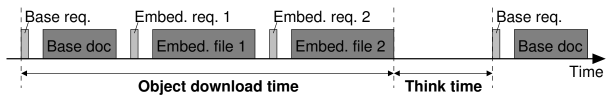


Fig. 8. HTTP model used by the "surge" Web traffic generator

longer idle times, if the congestion window validation [13] has reduced the window. Second, applications can explicitly trigger a Quick-Start request by setting a new socket option, either before or during the usage of the connection.

Quick-Start is particularly interesting for interactive applications that frequently exchange certain amounts of data and thus often suffer from the Slow-Start. In our experiments, we use two different types of interactive applications: First, we perform tests with simple client and server C programs that communicate as shown in Fig. 7, with variable amounts of data to be transferred. The measured server response time is averaged over five consecutive measurements.

Second, in order to study Web applications, we use a "lighttpd" Web server (version 1.4.18) and the "surge" Web traffic generator (version 1.00a) [14], which supports HTTP/1.0 and HTTP/1.1 requests, without or with pipelining. We use mainly the default configuration, i. e., there is a configurable number of users/threads that each send HTTP requests to the Web server. As performance metric we study the object download time, which is the total duration of loading a page consisting of a base document and potentially also embedded files (see Fig. 8). The measurement duration is one hour per sample. Concerning the object characteristics and request patterns, we first use the "surge default" Web model with rather small file sizes. However, recent studies [15] reveal that new Web applications frequently transfer objects of the order of 100 kByte (high-resolution images, complex database queries, 3D structures, ...). In order to reflect the traffic characteristics of such broadband interactive applications, we also use a second, customized "large files" workload model (see Tab. 1).

5 Experimental Evaluation

5.1 Quick-Start Validation

In the following, we discuss some results of systematic tests with the Quick-Start mechanism. Fig. 9 and 10 depict several traces of the traffic in download

Table 1. Used Web traffic model parameters (selection)

Parameter	**"Surge default" model**	**"Large files" model**
Ratio base/embedded/loners	0.30 / 0.38 / 0.32	0.30 / 0.38 / 0.32
File popularity	Zipf, 2000 files	Uniform, 2000 files
File size distribution	Mixed logn. and parato	Logn., $\mu = 11.92$, $\sigma = 1.0$
No. embed. files per object	Pareto, $\alpha = 1.245$, $k = 2$	Geometric, $1/p = 2$
User think time	Pareto, $\alpha = 1.4$, $k = 2$	Pareto, $\alpha = 1.4$, $k = 2$

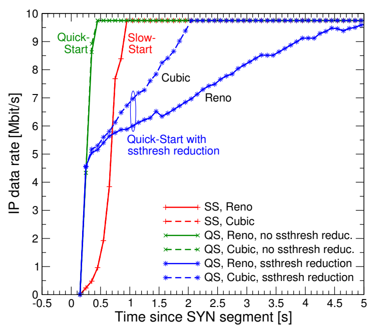


Fig. 9. Data rate trace ($RTT = 100\,\text{ms}$)

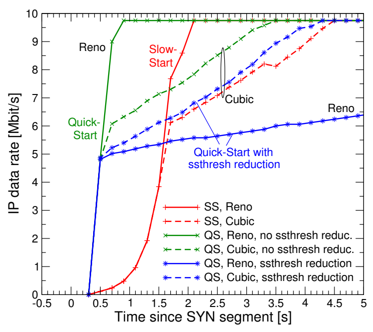


Fig. 10. Data rate trace ($RTT = 200\,\text{ms}$)

direction after the connection setup, comparing Slow-Start (SS) and Quick-Start (QS). The data rates have been obtained from a "tcpdump" trace by averaging the IP packet sizes within intervals of one RTT. We study the *Reno* congestion control, which implements [2], as well as the frequently used *Cubic* high-speed TCP variant [16]. In Fig. 9, the emulated minimum RTT is 100 ms. In this case a Slow-Start needs about one second to reach a data rate that completely fills the link, regardless of the congestion control variant. In contrast, the link can almost instantaneously be utilized with a Quick-Start request of $q = 5.12\,\text{Mbit/s}$, which is the largest possible request value smaller than the link capacity (see [1]).

The Quick-Start specification [1] leaves open whether to modify the Slow-Start threshold `ssthresh` [2] after a successful Quick-Start procedure. This variable is an estimation of the minimum available bandwidth, and it is challenging to define a reasonable initial value. Two alternative strategies could be:

1. No adaptation of `ssthresh` by the Quick-Start functions
2. Adaptation of `ssthresh` to the Quick-Start window (or some multiple of it)

The rational behind the latter strategy would be to use the information about the path characteristic obtained from the Quick-Start mechanisms.

Fig. 9 shows that a reduction of `ssthresh` can be disadvantageous compared to the first strategy, if the Quick-Start request rate q is smaller than the path capacity. In this case, the TCP sender enters the congestion avoidance phase before the link capacity is reached. Hence, it may require many seconds to send with full speed. This effect would be even more severe if a smaller Quick-Start request rate was used (e. g., $q = 2.56\,\text{Mbit/s}$). In the congestion avoidance phase, the *Cubic* variant outperforms *Reno* due to its higher aggressiveness.

For a higher RTT of 200 ms, the situation is more complicated, as shown in Fig. 10. In this case, the bandwidth-delay product of the emulated path corresponds to 167 packets, which is larger than the default initial `ssthresh` value of the *Cubic* algorithm in the used 2.6.20 kernel version[1]. With *Cubic*, the sender enters the connection avoidance state at a sending rate about 6 Mbit/s, and it

[1] Newer Linux kernels use a larger threshold for *Cubic* to overcome this problem.

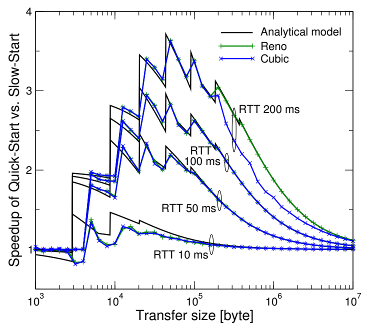


Fig. 11. Rel. performance improvement

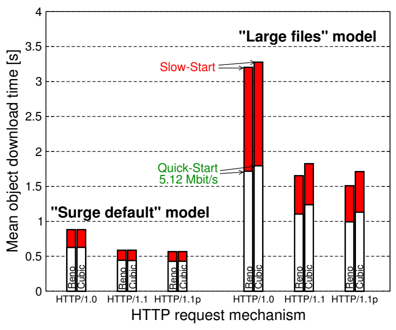


Fig. 12. Benefit for Web traffic (1 user)

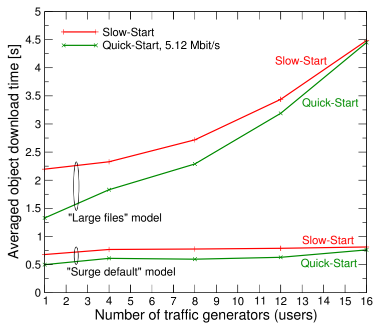
takes longer to fully utilize the path than *Reno*, which uses a very large default initial `ssthresh`. Again, a threshold reduction is detrimental: In particular the *Reno* congestion control requires then a long time to completely utilize the path, and the Slow-Start ends up to be faster. From this follows that `ssthresh` should not be decreased after a successful Quick-Start request. This is why we only use the "no adaptation" strategy in the following experiments.

5.2 Performance Benefit of Quick-Start

The measurements with our test programs can quantify how much the data transfer times could be improved by Quick-Start. Fig. 11 shows the relative speedup of data transfers $\eta = T_{\mathrm{SS}}/T_{\mathrm{QS}}$, where T is the server response time (cf. Fig. 7). An analytical expression for η has been derived in [4]. Our measurements confirm that Quick-Start can improve the response time by several hundred percent when the RTT is larger than 50 ms. Quick-Start is of particular benefit for transfer sizes between 10 kB and 1 MB, and the speedup is slightly larger for *Reno* because of the higher initial `ssthresh` value. For longer file transfers, the relative improvement is rather small because the Slow-Start is then only a transient effect. Very short transfers cannot be improved neither, since the initial congestion window is anyway large enough to send out such data immediately.

The comparison of the Linux measurement results with the analytical model from [4] reveals a close match, provided that the model takes into account the so-called "Quick ACKs" [17]. This mechanism of the Linux stack makes the Slow-Start more aggressive than foreseen by the TCP standards [2].

A result of our experiments with the Web server and the "surge" HTTP traffic generator is given in Fig. 12. Here, the minimum RTT is 200 ms, and the 10 Mbit/s link is used by one emulated user only. The mean object download time with the standard Slow-Start is always larger than with Quick-Start, independent of the HTTP protocol variant being used. However, the absolute difference is rather small for the "surge default" scenario. This is to be expected, since the mean size of an object (i. e., a Web page) is here only of the order of 17 kB. For larger file sizes, Quick-Start can significantly reduce the download duration.



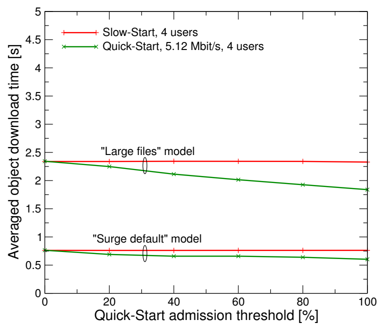


Fig. 13. Performance for increased load **Fig. 14.** Variation of the QS capacity

A speedup of more than 1 s can be achieved for HTTP/1.0 traffic, since here every file has to be transported over a new TCP connection suffering from the Slow-Start. But a significant improvement can also be observed for HTTP/1.1 persistent connections, both without and with request pipelining.

5.3 Impact of the Quick-Start Admission Control

At the Web server, the IP layer realizes the Quick-Start router functions, i. e., it monitors the utilization of the 10 Mbit/s link and processes the Quick-Start requests. They are approved if the resulting bandwidth, including the current traffic and recently approved request, is less than a certain threshold [1, 3]. If there are many requests in parallel, only a certain share will be approved. In Fig. 13, the number of users generating HTTP requests is successively increased, resulting in more Quick-Start requests and a higher link utilization. For simplicity we consider here the object download time averaged over six experiments with all combinations of *Reno*/*Cubic* and the three HTTP variants. As to be expected, download durations increase if the load gets larger. And the performance benefit of Quick-Start is also smaller, because less requests are approved and because the granted bandwidth is smaller. This illustrates that the Quick-Start mechanism mainly targets at underutilized links.

However, it is not necessary to allow Quick-Start to use the full link capacity: The diagram in Fig. 14 shows that application performance can also be improved if only a certain part of the capacity (e. g., only 50 %) is available for Quick-Start requests. This again confirms analytical predictions from [4].

6 Conclusion and Future Work

Quick-Start is an experimental TCP extension that allows to immediately utilize links in high-speed networks. This avoids the time-consuming Slow-Start, which is considered to be a limiting factor for broadband interactive applications. Quick-Start requires explicit router feedback, but it is rather lightweight compared to other related mechanisms that are proposed for a future Internet. This paper presents a Quick-Start implementation in the Linux stack and shows that the resulting overhead and performance cost is small. The impact of several parameters is studied by measurements, including both simple validation tests and experiments with HTTP traffic. Our measurements confirm the outcome of simulation studies, and they demonstrate that Quick-Start can significantly improve the performance of interactive applications communicating over long-distance wide area networks. Future work will also consider hardware-based Quick-Start implementations and compare Quick-Start to related congestion control schemes.

References

1. Floyd, S., Allman, M., Jain, A., Sarolahti, P.: Quick-Start for TCP and IP. IETF RFC 4782 (experimental) (January 2007)
2. Allman, M., Paxson, V., Stevens, W.: TCP congestion control. RFC 2581 (proposed standard) (April 1999)
3. Sarolahti, P., Allman, M., Floyd, S.: Determining an appropriate sending rate over an underutilized network path. Computer Networks **51**(7) (May 2007) 1815–1832
4. Scharf, M.: Performance analysis of the Quick-Start TCP extension. In: Proc. IEEE Broadnets. (September 2007)
5. Scharf, M., Strotbek, H.: Experiences with implementing Quick-Start in the Linux kernel. Presentation at IETF 69, Chicago, IL, USA (July 2007)
6. Liu, D., Allman, M., Jin, S., Wang, L.: Congestion control without a startup phase. In: Proc. PFLDnet2007. (February 2007)
7. Falk, A., Faber, T., Coe, E., Kapoor, A., Braden, B.: Experimental measurements of the eXplicit Control Protocol. Proc. PFLDnet2004 (February 2004)
8. Zhang, Y., Henderson, T.R.: An implementation and experimental study of the Explicit Control Protocol (XCP). In: Proc. IEEE Infocom 2005. (March 2005) 1037–1048
9. Dukkipati, N., Gibb, G., McKeown, N., Zhu, J.: Building a RCP (Rate Control Protocol) test network. In: Proc. IEEE HOTI. (August 2007)
10. Scharf, M., Floyd, S., Sarolahti, P.: Avoiding interactions of Quick-Start TCP and flow control. IETF Internet Draft, work in progress (July 2007)
11. Wehrle, K., Pählke, F., Ritter, H., Müller, D., Bechler, M.: The Linux Networking Architecture. Prentice Hall, Upper Saddle River, NJ, USA (2005)
12. Strotbek, H.: Design and implementation of a TCP extension in the Linux kernel. Diploma thesis (in German), University of Stuttgart, IKR (May 2007)
13. Handley, M., Padhye, J., Floyd, S.: TCP congestion window validation. IETF RFC 2861 (experimental) (June 2000)
14. Barford, P., Crovella, M.: Generating representative Web workloads for network and server performance evaluation. ACM SIGMETRICS Perform. Eval. Rev. **26**(1) (June 1998) 151–160
15. Schneider, F., Agarwal, S., Alpcan, T., Feldmann, A.: The new Web: Characterizing AJAX traffic. In: Proc. 9th International Conference on Passive and Active Netwo rk Measurement, Springer LNCS. (April 2008)
16. Rhee, I., Xu, L.: Cubic: A new TCP-friendly high-speed TCP variant. In: Proc. PFLDnet2005. (February 2005)
17. Sarolahti, P., Kuznetsov, A.: Congestion control in Linux TCP. In: Proc. USENIX Annual Technical Conference. (June 2002)