2000-05-12

TCP Control Groups: Aggregated Congestion Control for TCP (MA Thesis)

TCP Control Groups: An Aggregated Congestion Control for TCP

Thomas Gschwendter

Computer Science Department
Boston University
Boston, MA 02215
tgs@cs.bu.edu

Abstract

I present a framework for aggregated congestion management for TCP flows and show how to integrate it in an existing TCP protocol stack. I implemented a first version of my congestion management scheme in Linux and ns. Although my first results show that aggregated flows perform worse than individual flows, I am convinced that this is not a principle problem.

1 Introduction

Since TCP connections do not have any a priori knowledge of the available bandwidth of the path from the source host to the destination, they have to probe the available bandwidth. The congestion control and avoidance algorithm is responsible for that probing [4], [1]. If two or more TCP connections share the same bottleneck link, then this algorithm leads to competition for bandwidth between the connections. This competition might lead to a suboptimal link utilization. Although a lot of research has been done to make the congestion control algorithm "friendly" towards the competing connection(s) and, hereby, increase the link utilization, I believe that an explicit collaboration between such connection can further increase the link utilization and therefore the overall goodput. Furthermore, it can increase fairness. There is a lot of research which backs up my believe. For example, L. Qiu, Y. Zhang, and S. Keshav [8] showed by simulations, that there is a linear decrease in goodput as the number of connections increase. In addition, if the total number of packets that the bottleneck link and buffer can hold is smaller than the number of connections (small pipe case), then they found that TCP connections share the bandwidth very unfairly. Some connection get almost completely shut off in terms of goodput whereas others achieve a goodput that is considerably larger than zero. R. Morris [6] comes to similar results. TCP does not share a bottleneck fairly and efficiently, when the number of flows if higher than the number of packets in the delay-bandwidth product. With a large number of flows, the loss rate gets close to 50%. Morris identifies TCP's inability to have a congestion window smaller than one as one of the main causes for this problem. My approach will address this problem. A. Bestavros and I. Matta identify in their recent position paper [3] a "congestion control for collections of flows that share the same bottleneck" as a key component for a future Quality-of-Service architecture. They claim that aggregated congestion control leads to more stability, throughput, and fairness among flows. They support this claim by a brief analytic example and by simulations using ns [10].

I call my aggregated congestion control scheme *TCP Control Group*, *TCG* for short. A prerequisite for aggregating connection is that the connection share the source host, because there must be some fast and efficient means for information exchange and information sharing. This information exchange must be much faster than the network. Therefore the proposed aggregation is impossible between connections originated at different hosts. But I do not require that the TCP flows have the same destination host. Note that another pre-condition to build a TCG is the ability to gain some knowledge about the connections' bottleneck link. How to acquire this information is beyond the scope of this paper. Beside achieving better throughput and link utilization, more fairness and stability, another important goal of TCGs is to be "TCP friendly". But at the same time, TCP Control Groups do also not want to penalize connections that belong to a TCG over individual TCP connection. That is, n aggregated flows shall not be more aggressive than n individual flows, but the aggregated flows shall be able to get the same share of bandwidth of the bottleneck link in the same time as the individual flows. Or, in other words, if someone observes a network link and the only thing she can do is to record the time when a packet goes by, than it shall be impossible for her to tell whether the packets are sent by n individually controlled TCPs or n aggregated TCP. (At least it shall be impossible if the time scale of the observation is not several orders of magnitude smaller than the round trip time.)

The remainder of the paper is organized as follows. I give a general framework of TCP Control Groups and discuss details in chapter 2. In chapter 3 I discuss the Linux implementation and experimental results. The ns simulation are discussed in chapter 4. Related work is discussed in chapter 5 and I conclude in chapter 6. Finally, appendix A lists all added and changed files.

2 How TCP Control Groups Work

Overview. I will describe the general framework that incorporates TCP Control Groups into a TCP stack. Let's first briefly review how a packet is transmitted in a TCP stack (Figure 1). One of three events can trigger the transmission of a data packet, namely a send system call, an incoming acknowledgment, or a retransmission timeout. After one of these events has occurred, up to $\min(cwnd - packets\_out, rwnd - packets\_out)$ packets are transmitted, where *packets_out* is the number of unacknowledged packets, *cwnd* is the congestion window, and *rwnd* is the recently advertised receiver window. (Note that some implementations use sequence numbers as the unit for *cwnd* and *packets_out* and not number of packets.) In the case of a send system call these packets are provided by the system call itself, and in the case of an incoming ack or a retransmit timeout, these packets are taken from the transmit or retransmit queue respectively.

The packet transmission for a TCP flow belonging to a TCP Control Group differs in where and how the decision to transmit the packet is made. The flow control, i.e. how many packets to transmit based on *rwnd* and which packet to transmit next, stays with the flow itself, but a *group-global* instance, namely the TCG scheduler, takes over the congestion control (Figure 2). As with un-grouped TCP flows the three events send system call, incoming acknowledgment, and retransmission timeout can trigger the transmission of a packet. But if one of those events occur for a certain flow of the group, it does not necessarily mean that *this* flow can transmit the packet(s). The decision which flows are allowed to send data is up to the TCG scheduler. Upon the occurrence of one of the three events, the flow which received the event first updates its own, still individual state variables, then the shared state variables are updated, and after that the scheduler is called. The TCG scheduler examines *gcwnd* and *gpackets_out*. It then distributes the difference $gcwnd - gpackets\_out$ equally in Round-Robin fashion among the flows of the group. If a connection's flow control prohibits the connection from using the whole share that the scheduler assigned to the connection, then the scheduler will distribute the leftover share to other connections. There will only be a unused share after the scheduler is done, when every connection in the group was regulated down by its flow control. I have implemented TCGs in Linux and ns. I modified ns' TCP-Reno agent.

Detailed Discussion. In the following discussion, some terms are more related to an operating system environment than to ns, but it should be clear what the corresponding term or action is in ns. Every TCG is represented by a group control block *GCB*. The GCB contains the information listed below. Figure 3 shows the algorithm that is used by the Round-Robin scheduler. The TCG's connections are hold in cyclic linked list.
Fields of the group control block:

- *head*: Pointer to the head of the linked list, which contains the flows that belong to the TCG.
- *nof_flows*: Number of flows that belong to the group.
- *nof_active*: Number of active flows in the group. Active flows are flows that are participating in the aggregated congestion management. A flow can temporarily abandon its participation while it is in slowstart (see below).

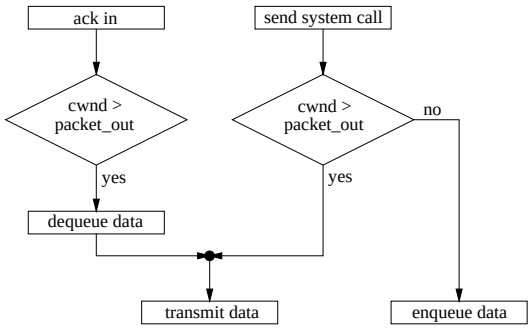

Figure 1: Typical TCP stack data flow

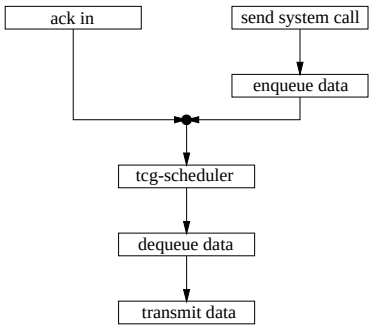

Figure 2: Data flow with TCG scheduler

- *current_flow*: Pointer to that flow, which will next get a share when the scheduler is called. After the scheduler has completed distributing the available share, this pointer is set to the first flow in the list which did not get the maximum share (see Figure 3).
- *gcwnd*: The aggregated congestion window of the group. *gcwnd* is updated according to the rules set forth below.
- *gpackets_out*: The number of unacknowledged packets of all flows belonging to the TCG.
- *gssthresh*: The aggregated slowstart threshold. It is not used in the current implementation.

Furthermore, the data structure, which represents a flow within the Linux kernel or in ns, was augmented by the following fields.

- *next*: Pointer to the next flow in the linked list.
- *active*: Set to true if the flow is active at the moment (see below), otherwise false.
- *share*: Contains the number of packets that a flow may send. After the scheduler has set this variable, it calls the function *tcg_transmit*, which then transmits the packets i.e., the flow hands over *share* packets to the IP-layer, if the flow control permits to do so. When tcg_transmit returns, the value of *share* is the part of the flow's share that the flow was not able to use due to a closed receiver window or because it had no more data to send, i.e. share is set to the difference of the original value of *share* and the number of sent packets. The scheduler will re-distribute this "leftover" to other flows.
- $inflation$: Records the amount of "artifical inflation" of *gcwnd* while the flow is in fast retransmit/fast recovery (see RFC2581 [1]). $inflation$ is initialize with zero.

I experimented with two different set of rules for updating the group congestion window. I implemented rule set (a) in both, Linux and ns, and rule set (b) only in ns. Unless otherwise noted, the scheduler is called on each of the listed events after the state variables are updated. In addition to *gcwnd*, *gpackets_out* is also updated. *gpackets_out* is always equal to sum of all unacknowledged packets over all flows of the group.

Rule set (a)

Event	Action
add connection to group	$gcwnd = gcwnd + cwnd$, where *cwnd* is congestion window of the connection to be added
send system call	—
new ack received *or* ≥ 4. duplicated ack received	$gcwnd = \max(1, gcwnd + cwnd - cwnd\_old)$, where *cwnd* is the flow's congestion window after the received ack is processed, and *cwnd_old* is the congestion window before the ack is processed
3. duplicated ack received	*gcwnd* is changed in the same way as with new ack received, but the presumably lost packet is immediately retransmitted and the scheduler is not called
retransmission timeout	$gcwnd = \max(1, gcwnd - cwnd\_old)$, where *cwnd_old* is the value of the flow's congestion window immediate before the timeout
remove connection from group	$gcwnd = \max(1, gcwnd - cwnd)$

```
tcg_scheduler(gcb) {
    if (gcb.gcwnd <= gcb.gpackets_out)
        return;

    /* current_flow will get a share first */
    flow = gcb.current_flow;

    /* Calculate the initial share */
    unused_share = gcb.gcwnd - gcb.gpackets_out;
    more = gcb.nof_active;
    newsctr = unused_share % more; /* To distribute the remainder equally */
    first = 1;

    while (more && unused_share) {
        share = unused_share / more;
        sremainder = unused_share % more;
        more = 0;

        for (i = 0; i < gcb.nof_flows && (share || sremainder); i++) {
            /* Give a share only to flows that are active and, in the first round,
               to every flow, but in subsequent rounds only to flows which used
               their whole share in the previous round */
            if (flow.active && (first || !flow.share)) {
                /* The new current_flow is the first flow, which does not get a share
                   of the remainder */
                if (newsctr) {
                    gcb.current_flow = flow.next;
                    newsctr--;
                }

                /* Calculate the share for the flow */
                if (sremainder) {
                    sremainder--;
                    flow.share = share + 1;
                }
                else
                    flow.share = share;

                /* Transmit up to flow.share packets.  After tcg_transmit returns,
                   flow.share will contain the amount of share, that the flow didn't use */
                tcg_transmit(flow);

                /* If the socket used the whole share it may be able to use more.  So,
                   if there is unused share at the end of the for-loop, try again */
                if (!flow.share)
                    more++;

                unused_share += flow.share;
            }
            flow = flow.next;
        }
        first = 0;
    }
    return;
}
```

Figure 3: The scheduling algorithm used by the TCG scheduler

Rule set (b)

Event	Action
add connection to group	$gcwnd = gcwnd + cwnd$, where $cwnd$ is congestion window of the connection to be added
send system call	—
new ack received *and* flow is not in fast recovery	$gcwnd = nof\_slowstart + (1 - nof\_slowstart) * nof\_active/gcwnd$, where $nof\_slowstart$ is the number of flows that is in slowstart
3. duplicated ack received	$gcwnd = gcwnd - gcwnd/(2 \cdot nof\_active)$; the presumably lost packet is immediately retransmitted, the scheduler is not called
$\geq$ 4. duplicated ack received	$gcwnd = gcwnd + 1$; $inflation = inflation + 1$
new ack received and flow is in fast recovery	$gcwnd = gcwnd - inflation$; $inflation = 0$;
retransmission timeout	$gcwnd = \max(1, gcwnd - gcwnd/nof\_active)$
remove connection from group	$gcwnd = \max(1, gcwnd - gcwnd/nof\_active)$

Slowstart deserves special attention. There are three obvious possibilities to determine when a flow is in slowstart: (a) The flow's congestion window and the flow's slowstart threshold is used to make the decision. (b) The flow's congestion window and the group's slowstart threshold is used. (c) The group's congestion window and the group's slowstart threshold is used. I will discuss each of the three possibilities.

(a) This version is used in the current implementation in Linux. Since the connection's individual congestion window does not necessarily reflect the portion of bandwidth which the connection occupies, the result might be twofold. On the one hand, consider the case where the congestion window is smaller than the slowstart threshold and the flow has more packets in the network than its own congestion window would allow. In particular, lets assume that the flow has as least as many packets in the network that the value of the congestion window which would correctly reflect the number of packets in flight is greater than the slowstart threshold. Then, to meet the "TCP friendly" target, the flow should perform congestion avoidance rather than slow start. Performing slowstart in this situation means that the flow is more aggressive than it would be without TCP Control Groups, because the flow can send two packets for *every* received ack. On the other hand, since the congestion window might be incorrect with respect to the number of packets in the network, the flow's slowstart threshold might also be an incorrect measure for the bandwidth that the flow can use without experiencing severe packet loss and it might enter congestion avoidance to early. To study the impact of slowstart on the overall performance, I provided a means which allows to exclude a flow from the group management temporarily. With this option enabled, a flow is removed from the group as soon as it enters slowstart and is again added to the group when it goes back to congestion avoidance.

(b) For the same reasons then (a), this might be more aggressive than regular TCP. But the group slowstart threshold should be a more accurate measure for the (fair) bandwidth available to the flow. A initial value for *gssthresh* could be one half of that value of *gcwnd*, where for the first time a flow of the group loses a packet. When flows in slowstart are excluded from the aggregated control and all flows are in slowstart, then *gssthresh* will not be updated since there is no flow which participates in the group management. The value of *gssthresh* might become stale. If the stale *gssthresh* is higher than a value that is a correct measure of the current conditions, then no flow will ever enter congestion avoidance and, therefore, will never again enter group management. A possible solution is that to enter congestion avoidance, a flow uses its own slowstart threshold and to enter slow start, it uses *gssthresh*. In addition, to prevent oscillation between congestion avoidance and slowstart, the first flow that enters congestion avoidance sets *gssthresh* to the value of its own slowstart threshold and subsequent flows add their slowstart threshold to *gssthresh*.

(c) Provided that there is a method to find the correct value of *gssthresh*, this version will be "TCP friendly". Since either *all* flows are or *no* flow is in slowstart, no flow that is in slow start will be able to "borrow" bandwidth form a flow that is in congestion avoidance and increase its share with this "borrowed" bandwidth in an over-aggressive way. One drawback of this method is that, in certain situation, the TCG increases its total bandwidth slower than the same number of individual connection would do. For example, consider a new connection that joins an existing TCG with many connection and assume that the TCG is in congestion avoidance. The new connection will have only a minor impact on the aggregated variables and, therefore, the TCG will stay in congestion avoidance. The new flow will be immediately in congestion avoidance. Hence, the gain of bandwidth of the group due to that new connection will be linear with time, whereas if the connection would be in slowstart, it would be exponential. A second drawback is that, although this method meets the "TCP friendly" criteria, it can be more aggressive in practice. All grouped flows perform slowstart synchronously and therefore all flows increase the bandwidth exponentially at the same time. In contrast to grouped flows, individual flows enter slowstart independent of each other. Therefore, it is unlikely that all individual flows will be in slowstart at the same time (unless the network is overloaded) and the bandwidth increase per unit time is less aggressive. Whether those drawbacks will result in a noticeable performance degradation is subject to further investigations. I did not evaluate this method.

3 Linux

API. The API consists of five functions hiding five new system calls. An application uses this function to create a TCG, add and remove connection to and from a TCG, and finally delete the TCG if it is no longer needed. The fifth function turns on a logging mechanism.

`XTcg CreateTcg(int type, int flags, int data);` Before the kernel can group together TCP flows, the application must call CreateTcg to set up the group control block within Kernel space. CreateTcg returns the handle (file descriptor) for the TCG that is used in the following API functions or NULL, if an error occurred. `type` must always be set to `C_TCG`. To exclude a flow that is in slowstart from the group management, set `flags` to `TCG_EXCLSLOWSTART`, otherwise set it to 0. `data` is not used in this version. The returned handle is a pointer into kernel space. The Application which uses this handle must not change it. A corrupted handle may crash the system or may lead to other serious and unpredictable effects. In a future, cleaner implementation, a file descriptor should be substituted for the kernel space pointer.

`int AddToTcg(XTcg tcg, int socket);` This function adds a TCP-socket to the (possibly empty) TCG. `tcg` is the handle of the TCG. `socket` is the socket file descriptor of the TCP connection that is added to the group. The function returns 0 if the connection was added successfully and -1 in the case of an error. A connection can not belong to two or more groups at the same time.

`int RemoveFromTcg(Xtcg tcg, int socket);` This function removes a TCP-socket from the TCG, i.e. the connection will be individually controlled after this function has completed. Again `tcg` is the handle of the TCG. The function returns 0 if the connection was removed successfully and -1 in the case of an error.

`int DeleteTcg(XTcg tcg);` This function deletes a TCG. `tcg` is the handle of the TCG. If there are TCP-connection, which belong to this TCG, this connection are individually controlled after this function call. The function returns the number of connections belonging to this TCG or -1 in the case of an error.

`int TcpSetLog(int socket, int flags);` This function turns logging of certain events on or off. Examples for such events are sending of a data packet or receiving an ack. For the complete list of all events see `tcp_group.h`. Logging can be turned on individually for one or more event(s). `socket` is the TCP-socket for which logging shall be turned on and `flags` is an event-constant as defined in `tcp_group.h`. To turn logging on for more than one event, use an ored list of event-constants. If an event occurs and logging is turned on for this event, then a sample is put into `/proc/ctcg`. A sample is a space-separated list of the following: time of occurrence (time is the kernel time, i.e. the time passed since system boot in 10 milliseconds), socket number (The socket number simply enumerates TCP sockets. It is different from the file descriptor.) , event number (as in `tcp_group.h`), slow start threshold, congestion window, number of packets in flight, group congestion window, number of packets in flight of the group, smoothed round-trip time. `/proc/ctcg` can be opened and read like a regular file using the standard open and read system calls. Since logging is implemented using Linux's sparse I/O model, more advanced file operations like lseek or the ANSI C file I/O functions like fscanf will not work. Once logged data are read, the read data will be removed, i.e. it is impossible to read the same data more than one time. Provided that there is sufficient system memory, up to 100000 samples are stored. If this number is reached, the kernel stops logging. Therefore, the user must read the data frequently enough to get a complete trace.

The API's prototypes are in `linux/tcp_group.h` and the code which maps the system calls proper onto the API functions is in `tcp_group.c`, i.e. the user must link `tcp_group.c` to her application to be able to use the new system calls.

The Server, the Client and Helper Programs. The program `tcg_server <port> <#bytes_to_transfer> <group> <exclude_slowstart>` is the server used in the experiments. The server will listening for connection requests on port `<port>`. After a request, it forks off a service process, generates `<#bytes_to_transfer>` dummy data and sends them to the client. The server closes the connection after it sent all data or after the client closed the connection, whatever occurs first. The `<group>` parameter is either `0` or `1`. `0` means the server uses regular, ungrouped TCP connections and `1` means the server puts all connections into one control group. To be able to compare grouped and individual TCP connections in an unpredictable and uncontrollable environment like the Internet, one can start one server for grouped connections listening at a certain port and a second server for individual connection at a different port on one host and perform group controlled and individually controlled data transmissions at the same time between one pair of hosts. If `<exclude_slowstart` is set to `1`, then a flow will be temporarily excluded from the group management while it is in slowstart, and if `<exclude_slowstart` is set to `0`, then a flow will all the time participate in the group management.

`tcg_client <server> <port> <#clients> <#experiments> <#bytes> <log_file>` is the client used in the experiments. It connects to the server with the name or IP-address `<server>` and port number `<port>`. It forks off `<#clients>` processes. Each process opens a connection and receives `<#bytes>` bytes of data. (Note: `<#bytes>` must always be smaller than the server's parameter `<#bytes_to_transfer>`. The latter is only kept for backwards compatibility.) After all connections closed their connections, `tcg_client` stores some statistical information about transfer times and throughput. The whole process is repeated `<#experiments>` times. When all experiments are done, it will write the information in the file `<log_file>`. The client can be compiled and executed on every UNIX platform, since it uses only standard C and UNIX libraries. However, it was only tested with Linux, Solaris and SunOS.

The program `tcgrlog [<file>]` reads the trace data from the file `/proc/ctcg` and prints them to `<file>`. If the filename is omitted, then it will print the data to standard out. There are several Perl scripts to process the trace data and the log file generated by the client. Every script includes a brief description about itself. Please check there for more information.

Implementation in the Kernel. Almost all additions to the kernel are in two files, `tcp_group_sys.c` and `tcp_group.h`. Existing files were only changed when it was absolutely necessary. There are several reasons which made those changes necessary: to add the new system calls, to call the group scheduler whenever the application executed a send system call,

	Grouped Flows	Individual Flows
Average Goodput (bytes/s)	41246	83150
Average Latency (s)	199	110
Max. Latency (s)	201	136
Total Number of Timeouts	83	57

Table 1: Results of one experiment with 10 grouped and 10 individual flows

to update the state variables of the group and call the group scheduler where a incoming ack or a timeout is processed in the original TCP stack, and to facilitate logging. Appendix A lists all changed and new files and their location within the kernel source file tree.

Results and Problems. The server with the modified Linux kernel was located at Boston University. I did experiments with clients on a host at the University of Southern California and with clients on a host in Munich, Germany with similar results. As mentioned above, I implemented rule set (a) to update the aggregated congestion window. To achieve comparable results, every individual experiment had as many grouped flows as un-grouped flows. I did experiments starting with 4 flows total up to 100 flows. My experiments have shown, that, in general, grouped connections perform worse than individual connections. Only if the number of group connections is smaller than 5 then they do usually better. With increasing numbers of grouped connection, there is very roughly a linear decrease in Goodput. This linear decrease might be an indication that if one connection has to back-off, the other connections will also (unnecessarily) suffer from that back-off. I observed that grouped connection experience a greater number of timeouts than individual connections. This might be an indication that group flows are more aggressive than ungrouped flows. Excluding that connections which are in slowstart does not considerable change the results. As an example, table 1 shows the result of one experiment with 20 flows total. I explored further reasons with ns simulations. A detailed discussion about possible reasons is in chapter 4.

4 Network Simulator ns

Changes to ns. Due to the object oriented design of ns, it was possible to extend ns' functionality only by inheriting classes and overwriting certain member functions. The original ns source code is kept (almost) unchanged. I had to change the file `tcp.h`, because some function that I overwrote were not declared virtual. The second changed file is `ns-default.tcl`. It includes now initialization values for some new variables. There is the new protocol agents `Agent/Tcp/Reno/Group` which base class is ns' Reno TCP agent, and there is a new class for the group control block which is derived from the class `TclObject`. Its Tcl object name is `TCG`. To create an instance of a group control block in a Tcl script use the command `set $tcg [new TCG]`, and to create an instance of the new protocol agent us the commend `set $atrg [new Agent/Tcp/Reno/Group]`. A group Reno TCP agent behaves exactly like the original Reno TCP agent as long as it is not added to a group. Following is a list of commands of the two new classes:

`$tcg set-mode 0|1`; `1` means that flows in slowstart are temporarily excluded from the group management and `0` means that flows participate in the group management as long as they belong to a group. The mode must not be changed after a flow was added to the group. The default value is `0`.

`$atrg add-to-tcg $tcg`; This command adds the group Reno TCP connection `$atrg` to the group `$tcg`.

`$atrg remove-from-tcg $tcg`; This command removes the group Reno TCP connection `$atrg` from the group `$tcg`.

`$atrg tcg-trace [<filename>]`; This command enables a additional trace feature. The trace data are printed into the file `<filename>`. If `<filename>` is omitted, then a previously opened trace file is closed. On linux, more than one agent can open the same file such that the trace data of several agent will end up in the same file. This might not work correct on other platforms depending on the operating system's I/O model. Please see the function `GroupRenoTcpAgent::tcg_trace` in `tcp-tcg.cc` for the format of the trace file.

Simulation with ns. I used rule set (a) and (b) in ns. Rule set (a) produces similar results as Linux. Rule set (b) shows improvement, but still not satisfying results. In the current version, rule set (b) is implemented. I show first the results of two simulations with rule set (a), each experiment is with 2 grouped TCP flows and 2 individual TCP flows. The difference is that there is no cross traffic in the first simulation, whereas in the second simulation bursty on-off

cross traffic is added. The topology for the used network is a simple one-link-topology. The link is used by all flows. Figures 4 to 7 show the results. As can be seen from the figures, the grouped flows perform worse than the un-grouped. My initial explanation was the following: As can be seen from Figures 5 and 7, there are times where the congestion windows diverge significantly. When the flow with the bigger congestion window will detect a packet loss and therefore will decrease its congestion window, then, in the case of groups, both flows have to suffer from the relatively big back off. Furthermore, since the big congestion window does not correctly reflect the portion of bandwidth used by the flow, the back off might be too big. The too big back off results from the way the group congestion window is updated due to rule set (a). Remember that *gcwnd* changes (approximately) by the same amount the individual congestion window changes and the scheduler distributes the available bandwidth equally and independent of the flow's own congestion window. Rule set (b) should cure this wrong behavior, since the change of *gcwnd* due to a lost packet is closer to the number of packets a flow has in the network.

I will now show 3 simulations with rule set (b) as an example. I use again a one link topology. Each simulation has 4 flows under group control and 4 individual flows. The traffic generator is ns' constant bit rate application. There are 10 on-off cross traffic flows. The bit rates during on-time is 500 kbit/s. The on and off times follow a Pareto distribution with shape parameter 1.5. The link delay is set to 10 ms. All these parameters are constant in the three simulations, only the queue length varies. Figure 8 shows the sequence number plot of the simulation with a queue length of 20 packets, Figure 9 with a queue length of 10 packets, and figure 10 with a queue length of 5 packets. Tables 2, 3 and, 4 show further results. Simulations with other parameters led to comparable results. It can be clearly seen from the tables and the figures that the heavier the congestion, the worse grouped flows perform compared to individual flows. I identified two possible reasons for that behavior:
(i) Exponential back-off of retransmit timeout is never cleared. This effect can be seen in plot 10. The sequence numbers of grouped flow 1 do not increase between approximately 6.7 and 11.3. The reason is that all four grouped flows have lost several packets, fast retransmit does not work because there are to many packets dropped, and all four flows are waiting to timeout. The exponential back-off is unreasonable hight, and therefore it takes to long to timeout. How does this too high back-off build up? Let's consider the case where one flow of the group does not get several packets acknowledged. If this flow had been on its own, then it would not have been able to send more packets until a timeout. But in a group it can continue to send packets provided that other flows in the group are receiving acks. Therefore, the difference between the highest acknowledged sequence number and the highest unacknowledged sequence number may get big. ns resets the exponential back-off only if a new packet was transmitted, i.e. a packet that advances the highest unacknowledged sequence number, *and subsequently* this packet or a packet with a higher sequence number is acknowledged. Because of the big difference of the acknowledged and the unacknowledged sequence numbers, many timeouts may occur until the back-off is cleared. This leads to much higher back-off value than it would be with individual flows. (ii) The group congestion window does not decrease fast enough in certain situations (see figure 11, the group congestion window is sometimes much higher than the sum of the individual congestion windows), therefore grouped flows do not adopt fast enough to the network. Let's consider the following two example, one with grouped flows and one with regular TCP flows. Assume that we have 10 regular flows and that the sum of their congestion windows is 40. Assume that all flows timeout at approximately the same time. The sum of their congestion windows will now be four, since ever flow reduced its window to one. Consider the same situation with 10 grouped flows. Since their timeouts are processed one after another, we can apply the corresponding formula of rule set (b) recursively to the starting value 40. The result is 13.9. Thus, the resulting congestion window is much higher than in the case with regular TCP. Furthermore, it can be seen from figure 10 that under heavy congestion, some flows are able to advance their sequence number faster than others, i.e. some flows receive a higher throughput and the goal fairness is not achieved. This is due to a mechanism that tries to minimize unnecessary retransmissions after timeouts. When a timeout occurs, the flow is only allowed to retransmit the packet that timed out. Only after this packet is acknowledged the flow can again participate in the Round-Robin scheduling. With out this mechanism a flow would retransmit successive packets which might not be lost.

5 Related Work

RFC 2140 [9] suggests that new TCP connections can improve their startup performance by getting information from older, probably no longer existing connections with the same source and destination host. Specifically, the new connection can simply copy the cached values of the RTT, the congestion window and the MSS of the no longer existing connection. Furthermore, if two or more connections with the same source and destination host exist at the same time, they can share one RTT and MSS state variable. Every connection will update the variable whenever it has new information. The authors also discuss how a new connection can initialize its congestion window by making use of the congestion windows of concurrent connections, however they do not consider sharing of the congestion window information while connections are in progress. They admit that the assumptions on which their approach is based can be incorrect and that further

research is needed. There are no analytic or experimental results or other indication what performance gain to expect from deploying the proposed scheme.

V. Padmanabhan's and R. Katz's [7] TCP Fast Start uses ideas similar to those in RFC2140. In particular, TCP Fast Start caches the congestion window, the slowstart threshold and the round trip time. A new connection uses this values as initial values. Since a high initial congestion window would lead to large, undesired burst, timers spread out the packets over one RTT during Fast Start. The authors augmented the algorithm with some heuristics to detect stale cached information fast and, if the information is indeed stale, fall back to regular slowstart. To ensure that the more aggressive Fast Start algorithm will not hurt regular TCP traffic, packets sent during Fast Start are tagged and routers drop tagged packets with higher priority than un-tagged packets.

H. Balakrishnan, V. Padmanabhan, et al. [2] propose an aggregated congestion control called TCP-Int. TCP-Int uses, as TCP Control Groups do, a common congestion window variable and the sum of the the number of packets in flight over all connection as the base for the decision how many packets can be sent. The main difference to TCP control groups is that the commonly controlled connection must not only have the same source, but also the same destination host. On the one hand, this restriction allows them to implement an integrated loss recovery algorithm similar to fast retransmit/fast recovery of standard TCP but on the other the number of connection that TCP-Int can aggregated might be considerably smaller than the number of connection TCP Control Groups can aggregate. In this sense, TCP Control Groups could be seen as the generalization of TCP-Int to connection which share the same bottleneck router but do not necessarily need to share the same destination host. Furthermore, when a packet loss is detected, TCP-Int decreases the aggregated congestion window in the same way as a single TCP connection would decrease its congestion window. That means that in the case of a packet loss of one connection of the aggregate, the back off is bigger than the overall back off of the same number of individual TCP connection would be. Therefore, TCP-Int can be seen as being more "TCP friendly" than TCP itself.

H. T. Kung and S. Y. Wang [5] present a different method of aggregating TCP flows, TCP Trunking. A TCP trunk is implemented on layer three of the protocol stack on top of a layer two virtual path or an MPLS switched path.

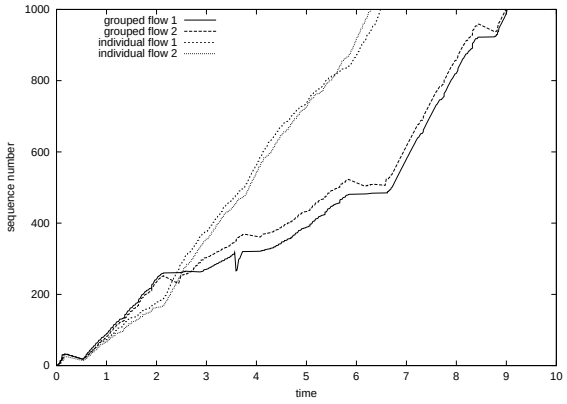

Figure 4: Sequence number plot, without cross traffic, slowstart phase excluded

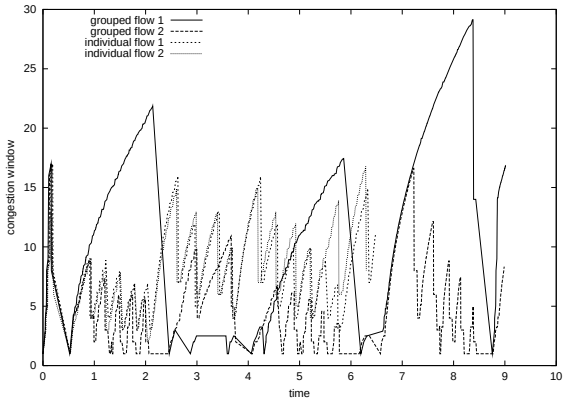

Figure 5: Individual congestion windows, without cross traffic, slowstart phase excluded

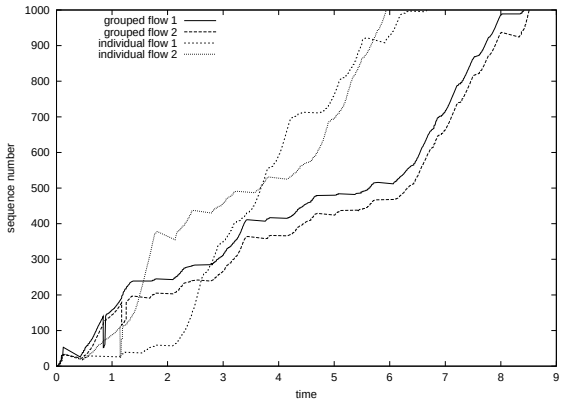

Figure 6: Sequence number plot, with cross traffic, slowstart phase excluded

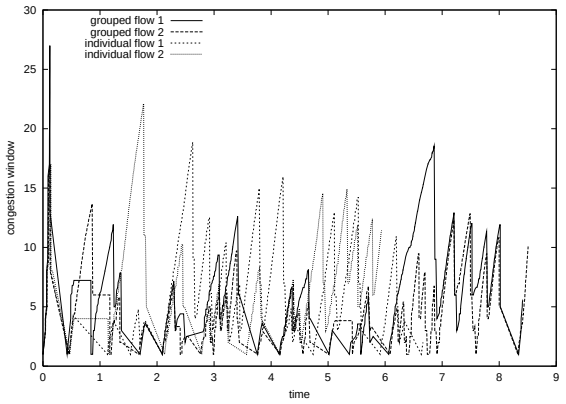

Figure 7: Individual congestion windows, with cross traffic, slowstart phase excluded

Typically, a TCP trunk connects two network domains. Therefore, the TCP Trunking algorithms are implemented in gateways rather then in end-hosts. Both, the sending and the receiving gateway must implement trunking. A TCP trunk carries a number of user flows, which are TCP or UDP flows between end-hosts. A TCP trunk has one or more management TCP flows, which implement the standard TCP congestion control algorithm. Those flows do not carry payload. Their task is to determine the rate at which user flows can send data. Every management packet is followed by a certain number of user bytes. In this way, the management TCP controls the rate at which user flows can send data. The gateway that implements the trunk might queue and probably drop user data. With certain modification of the routers on the path from the trunk sender to the trunk receiver, the TCP trunk can guarantee lossless delivery and/or minimum bandwidth. The authors show with simulations that trunks share the available bandwidth fairly according to their allocated proportion and get their allocated guaranteed minimum bandwidth. By putting UDP flows and TCP flows into separated trunks, UDP flows, which are not "TCP friendly", can no longer hurt the TCP flows. Furthermore, they showed that trunks help short-living TCP flows to receive an at least fairer share when competing with long-living TCP flows as the would be able to achieve as regular, un-trunked TCP flows.

6 Conclusion

I believe that an aggregated congestion management is a key feature for future Quality-of-Service architectures. I developed a general framework for an aggregated congestion manager for TCP and showed how it can be integrated in an existing TCP protocol stack. With my Linux and ns implementation, I provide tools to explore many aspects. Although my current results do not show an improvement, I believe that further research refinement of the suggested method will lead to beneficial behavior.

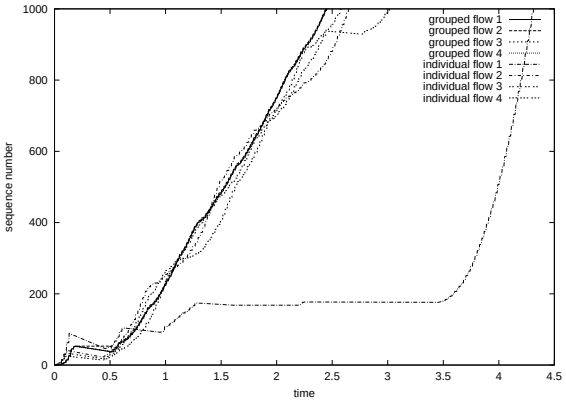

Figure 8: Sequence number plot, rule set (b), with cross traffic, queue length 20 packets

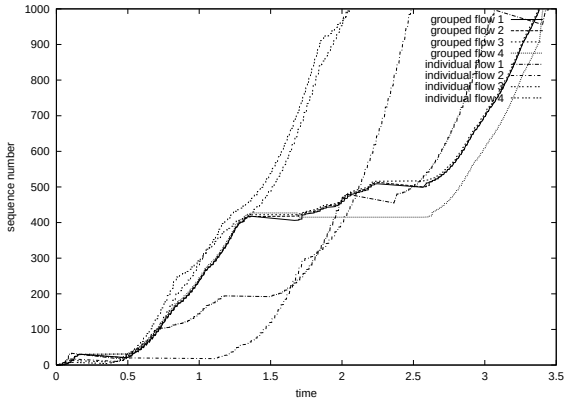

Figure 9: Sequence number plot, rule set (b), with cross traffic, queue length 10 packets

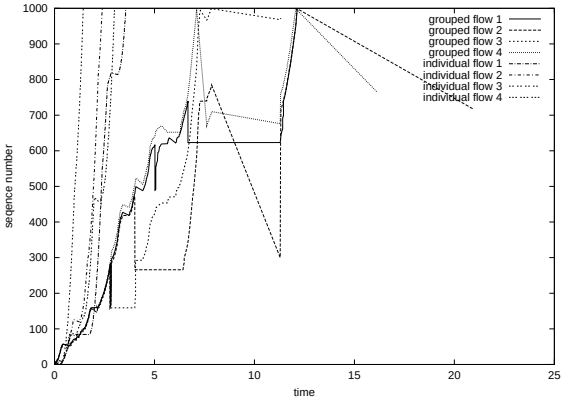

Figure 10: Sequence number plot, rule set (b), with cross traffic, queue length 5 packets

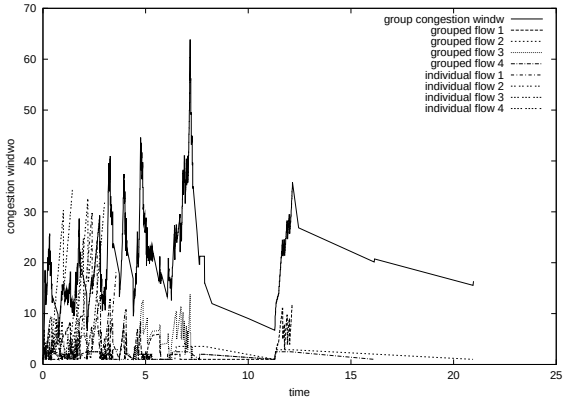

Figure 11: Congestion windows, rule set (b), with cross traffic, queue length 5 packets

	Grouped Flows	Individual Flows
Retransmitted packets	90	126
Timeouts	4	9
3 dup acks	50	34
Packets dropped	60	85

Table 2: ns simulation with queue length 20 packets

	Grouped Flows	Individual Flows
Retransmitted packets	88	58
Timeouts	8	8
3 dup acks	60	26
Packets dropped	76	49

Table 3: ns simulation with queue length 10 packets

	Grouped Flows	Individual Flows
Retransmitted packets	843	44
Timeouts	30	10
3 dup acks	136	24
Packets dropped	272	42

Table 4: ns simulation with queue length 5 packets

References

[1] M. Allman, V. Paxson, and W. Stevens. *TCP Congestion Control*. IETF, April 1999. RFC 2581.

[2] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. Tcp behavior of a busy internet server: Analysis and improvements. In *Proceedings of Infocom*, March 1998.

[3] A. Bestavros and I. Matta. Qos controllers for the internet. Technical report, Boston University, 2000.

[4] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM*, August 88.

[5] H. T. Kung and S. Y. Wang. Tcp trunking: Design, implementation and performance. Technical report, Harvard University, 1999.

[6] R. Morris. Tcp behavior with many flows. In *Proceedings of ICNP*, October 1997.

[7] V. N. Padmanabhan and R. H. Katz. Tcp fast start: A technique for speeding up web transfers. In *Proceedings of Globecom*, November 1998.

[8] L. Qiu, Y. Zhang, and S. Keshav. On individual and aggregated tcp performance. Technical report, Cornell University, May 1999.

[9] J. Touch. *TCP Control Block Interdependence*. IETF, April 1997. RFC 2140.

[10] UCB, LBLN, and VINT. Network Simulator - ns (version 2), 1999. http://www-mash.cs.berkeley.edu/ns/.

A Location of Files

There are three file archives which contain all files. `tcg_linux.tar` contains the whole Linux 2.2.10 kernel including the new files. `tcg.tar` contains the user space programs and scripts.

`tcg_ns.tar` contains the two files that were added to ns, the ns files that were changed, the Tcl script that were used for the experiments, and several Perl scripts to evaluate the trace files. Since the installation of ns is platform specific, I did not include the Makefile. After the added and changed files were copied into the appropriate ns directories, the Makefile must be changed such that the file `tcp-tcg.cc` is compiled and linked. The ns package I used was ns-allinone-2.1b6.

The following tables list all add and modified Linux kernel files, user space program files and scripts, and all added and modified ns files and scripts.

Linux Kernel Files

File	A/C	Description
linux/arch/i386/entry.S	C	New system call entry points.
include/asm-i386/unistd.h	C	New system call numbers.
net/ipv4/tcp_group_sys.c	A	Contains all added functions like the TCG scheduler, code to update the group state variables, new system calls, functions for logging, etc.
include/linux/tcp_group.h	A	API prototypes and constants for applications as well as prototypes, inline functions, constants, macros, and data structures used within the kernel.
include/net/sock.h	C	Extensions to the socket data structure and to the socket locking/synchronization mechanism. The TCG scheduler is called if a previous call could not be completed due to an active lock.
net/ipv4/tcp_output.c	C	Update of group state variables after transmitting or retransmitting packets. The TCG scheduler calls functions of this module in order to transmit or retransmit packets.
net/ipv4/tcp_input.c	C	Update of group state variables after incoming ack. Call of TCG scheduler, since an ack might open the congestion window or the advertised window. Since acks are processed in an interrupt level (bottom half), the scheduler is not called directly. Rather, the kernel checks whether the scheduler is locked and if this is the case, it postpones the execution of the scheduler. Then, the scheduler is called as soon as the lock is released. Furthermore, if flows do not participate in the group management during slowstart, then flows are added to and removed from the group as they leave or enter slowstart.
net/ipv4/tcp_timer.c	C	Update group state variables after timeout.
include/net/tcp.h	C	TCG scheduler is called to send pending packets after a send system call.
net/ipv4/tcp_ipv4.c	C	Update of group state variables after ICMP error.
fs/proc/root.c	C	Create file `/proc/ctcg` in proc file system.

Linux User Space Programs and Scripts

File	Description
tcg/tcp_group.c	The code that wraps the system calls proper into the API.
tcg/tcg_server.c	The source code for the server used in the experiments. To compile, link against tcp_group.o, i.e. `gcc -o tcg_server tcg_server.c tcp_group.c`.
tcg/tcg_client.c	The source code for the client used in the experiments
tcg/tcgrlog.c	Program to read the trace data from `/proc/ctcg`.
tcg/klog2c.pl	Splits the read trace data (trace file) by connections, i.e. if there are x connections traced in the file, then their will be x files, each containing the data for one connection.
tcg/klog2e.pl	Splits the trace file by event, i.e. there will be one file for each event.
tcg/klog2g.pl	Splits the trace file by group membership, i.e. there will be one file that contains all connection which belonged to a group and another file that contains the connection which did not belong to group.
tcg/klogstat.pl	Computes some statistics from the trace data.
tcg/klog2col.pl	Extracts 2 column from the trace file. gnuplot can plot the resulting file. The first column is always the time and the second column is given as the first command line argument.
tcg/tcglog2s.pl	Computes some statistics from data collected at the client

ns Files

File	A/C	Description
ns/ns/tcp-tcg.cc	A	Implements the TCP Reno group agent class and the group control block class
ns/ns/tcp-tcg.h	A	Header file for `tcg-tcg.cc`
ns/ns/tcp.h	C	Changed some member function declarations
ns/ns/tcl/lib/ns-default.tcl	C	added intimal values for new variables
ns/tcg/tcg-1.tcl	A	Tcl script used for the simulations in this paper
ns/tcg/sim.pl	A	Perl script that calls `tcg-1.tcl` to start the simulation. After the simulation is done, it evaluates the trace files and calls further Perl scripts to generate files for gnuplot. It assumes that tcg-1.tcl will put its trace files into `ns/tcg/log/tmp` and it assumes certain file names. Leave `tcg-1.tcl` unchanged with respect to the trace file names.
ns/tcg/plotdata.pl	A	Generates files for sequence number plot and congestion window plot that can be plotted using gnuplot. Called by `sim.pl`.
ns/tcg/gcwndpout.pl	A	Generates files for group congestion window plot and packets in flight plot. Called by `sim.pl`.
ns/tcg/interpt.pl	A	Generate file for inter packet gap histogram. Called by `sim.pl`
ns/tcg/plot.pl	A	Interactive script to plot data generated with above scripts using gnuplot.