

Implementation Issues of the Autotuning Fair Share Algorithm

Jeffrey Semke
Pittsburgh Supercomputing Center
Carnegie Mellon University

May 15, 2000

PSC Technical Report # CMU-PSC-TR-2000-0002

Introduction

There are two main components in the sender-side implementation of TCP Autotuning socket buffers [SMM98]. The first component determines what socket buffer size a connection desires based on its congestion window, which reflects this connection's round trip time and its share of the available bandwidth. The second component is the algorithm that is used to determine a connection's fair share of the memory resources, since not all connections may be able to obtain the buffer sizes they desire. This document describes problems experienced in implementing the second component, the **max-min fair share algorithm**.

In this paper, the terms **allocate** and **reserve** are used to describe the *conceptual* assigning of memory to connections by the fair share algorithms. In the actual TCP Autotuning implementation, the fair share algorithms set *upper limits* of memory usage only. Actual memory allocation is independent of the algorithm and is performed on an as-needed basis.

A Problematic Implementation of the Max-min Fair Share Algorithm

The idea of using the max-min fair share algorithm to determine resource allocations for TCP Autotuning was inspired by a paper named "Routing high-bandwidth traffic in max-min fair share networks" [MSZ96], which described the use of the max-min fair share algorithm to allocate data rates to flows traversing links of a network.

In the first implementation of TCP Autotuning [SMM98], each connection determined its desired socket buffer size by multiplying its congestion window size appropriately. A pool of memory was made available for use in socket buffers for connections. Connections that desired less buffer space than the *fair share* reserved their desired amount of space. All the remaining memory in the pool was divided equally among the connections that desired more than the fair share. This equal share was defined as the new *fair share*, for use in the next periodic allocation of buffer space.

Each periodic allocation round involved three steps. The first step was to count the number of *small* connections, the number of *large* connections, and the sum of the buffer space desired by the small connections. The second step was to calculate the new *fair share*.

$$\text{new fair share} = \frac{\text{total pool} - \text{sum of buffer space of small conns}}{\max(\text{number of large conns}, 1)}$$

The third step was to allocate the $\min(\text{new fair share}, \text{desired amount})$ of buffer space to each of the connections. The first step was implemented as one loop through all of the connections, while the third step was implemented as another (non-overlapping) loop.

The *fair share* would converge as long as each connection used an insignificant fraction of the memory resources. However, an oscillation could result when there are only a few connections, as described in the following example.

Let C be a particular small connection. As C 's desires grow to exceed the *fair share*, it becomes a large connection, donating the buffer space it had reserved into the shared pool. Now the larger pool is divided up among the large connections (of which there is one more now), and the amount each large connection receives is the new *fair share*.

Because the pool is larger than it was before C donated its buffer space into the pool, the new *fair share* may be larger too. If the new *fair share* is larger than C 's desires, then C drops back to being a small connection in the next periodic allocation round, taking its desired buffer space out of the shared pool and keeping it for its own use. The smaller pool results in a smaller *fair share*, and an oscillation may begin.

The problem of buffer space thrashing back and forth between a connection's reserved buffer space and the shared pool was the result of an implementation detail.

The Corrected Fair Share Implementation

The corrected fair share implementation for NetBSD is available from PSC's web site [\[PSC98\]](#) by downloading an implementation newer than September 1999. The corrected implementation begins each periodic allocation round by assigning zero space to each connection (deallocating the entire buffer pool), then adding progressively-smaller "slices" to each connection's allotment until either all connections are satisfied, or the pool is empty.

Every connection gets up to $1/N$ th of the unallocated pool, taking no more than it needs. If a connection needs less than its $1/N$ th portion, the remainder of the portion is left in the pool. After giving each connection a "slice", the remaining unallocated pool is divided up among the unsatisfied connections, continuing in this manner until either the pool is empty or all connections are satisfied. More precisely:

1. Let P = the total size of the pool, allocated and unallocated parts.
Let $D(i)$ = the amount of buffer space desired by connection i .
Let $B(i)$ = the amount of buffer space allotted to connection i so far (zero initially).
2. Let U be the set of all connections, i , for which $B(i) < D(i)$.
Let N = the number of connections in U .
Let the *current share* = $(P - \text{sum}(B(i)) \text{ for all } i) / N$
3. For all i in U ,
 - if $D(i) < B(i) + \text{current share}$ then let $B(i) = D(i)$
 - otherwise, increase $B(i)$ by the *current share*
4. Repeat from step 2 until U is empty, or the *current share* falls to a certain threshold (which may be zero).

For efficiency of processing, the threshold in the final step can be set to an arbitrary value to save processing time by sacrificing memory efficiency. (For instance, don't bother trying to allocate "slices" smaller than 512 bytes, since the additional buffer space won't make much difference, but allocating it will cost more iterations.)

This corrected implementation matches the algorithm described by Ma, et al. [\[MSZ96\]](#) more closely than the problematic implementation.

Comparison of the Implementations

The corrected implementation does not exhibit the "thrashing" effect that causes oscillations in the problematic implementation, since connections do not reserve or donate their entire buffers to the shared pool all at once.

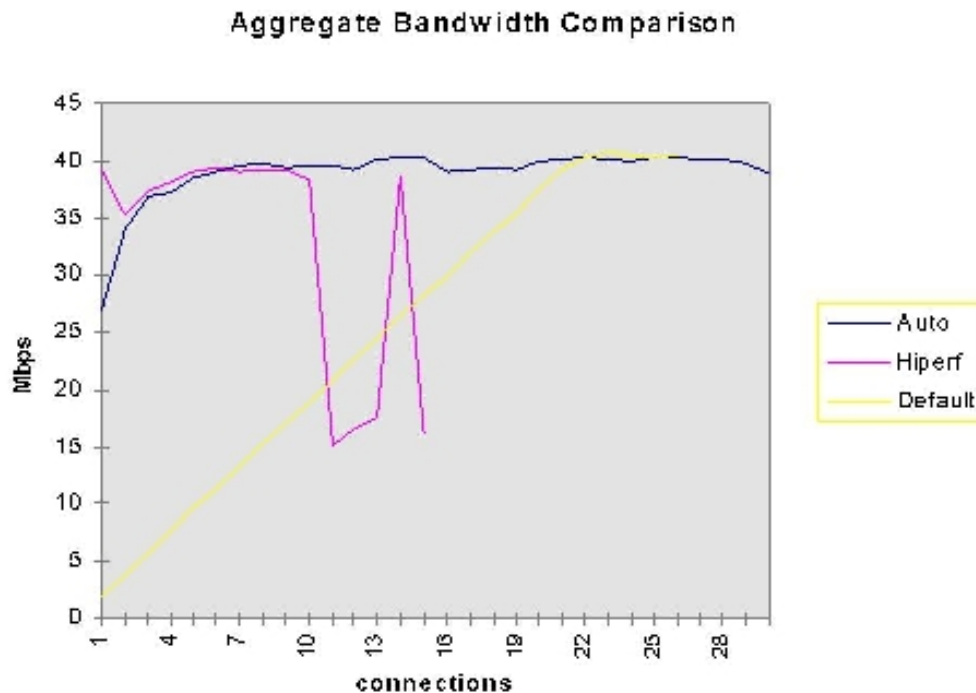
The three-phase allocation process used by the problematic implementation can result in the values measured in the first phase (first loop) being invalidated when the third phase is reached. This implementation uses the outdated *fair share* in the first phase when it counts the number of small and large connections and sums the desired buffer space of the small connections. These values are used to calculate the new *fair share* in the second phase. When allocations are made in the third phase (second loop), the number of small connections measured in the first phase may be wrong if the fair share has changed. The fair share then oscillates between two values.

The corrected implementation allocates slices throughout the process, rather than having explicit phases. All connections are

treated the same, without basing calculations on how many connections have been classified as "large" or "small". A single loop handles the entire allocation round.

Relevance to previous work

A question was raised after the publication of the SIGCOMM98 Autotuning paper [SMM98] concerning the "Aggregate Bandwidth comparison" graphs. The graphs show the aggregate bandwidth achieved by a number of concurrent connections that all use a particular socket buffer allocation scheme: fixed buffers of small "default" size, fixed buffers of large "hiperf" size, and dynamically self-sizing "auto" buffers. In the graph, the "default" buffers are not able to make use of all of the available bandwidth until a large number of them are running concurrently. The "hiperf"-tuned connections are able to fill the link, but starve the sender for memory when too many of them are running concurrently. It was not noticed to be relevant until after publication that when only a few "auto" connections were running, they were not able to achieve as much aggregate bandwidth as the "hiperf" connections. The graph below was taken from the SIGCOMM98 paper (and was modified for the SIGCOMM presentation). On the left side of the graph, it can be seen that the autotuning performance dips below the "hiperf" connections for no apparent reason.



Following publication of the paper, the kernel monitoring tool was enhanced [SEM00], enabling the discovery of the oscillation of the first implementation. The author believes at this time that the lower performance of the few "auto" connections is likely to result from the oscillating fair share of the first implementation. Further testing is required to validate this belief, and is expected to be conducted as a by-product of upcoming projects.

Conclusions

An implementation issue of the **max-min fair share algorithm** was been described. The original implementation of the algorithm for TCP Autotuning was found to be incorrect. The source of the error and its effects were addressed, while a corrected implementation which solves the problem of oscillating allocations was put forth.

References

- [MSZ96] Qingming Ma, Peter Steenkiste, and Hui Zhang. "Routing high-bandwidth traffic in max-min fair share networks." *ACM Sigcomm '96/Computer Communications Review*, Volume 26, August 1996.
- [PSC98] "Automatic TCP Buffer Tuning Research" web page, Pittsburgh Supercomputing Center, 1998. Available at:
<http://www.psc.edu/networking/auto.html>
- [SEM00] J. Semke. "PSC TCP Kernel Monitor," Technical Report #CMU-PSC-TR-2000-0001, Pittsburgh Supercomputing Center, Carnegie Mellon University, May 2000. Available at:
http://www.psc.edu/publications/tech_reports/pscmonitor/pscmonitor.html
- [SMM98] J. Semke, J. Mahdavi, and M. Mathis. "[Automatic TCP Buffer Tuning](#)," *ACM Sigcomm '98/Computer Communication Review*, Volume 28, Number 4, October 1998.