ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220725214

DTL: Dynamic Transport Library for Peer-to-Peer Applications

Conference Paper · January 2012

DOI: 10.1007/978-3-642-25959-3_32 · Source: DBLP

CITATIONS
6

READS
108

4 authors:

Riccardo Reale
Swedish Institute of Computer Science
3 PUBLICATIONS **16** CITATIONS
SEE PROFILE

Roberto Roverso
Peerialism AB
19 PUBLICATIONS **114** CITATIONS
SEE PROFILE

Sameh El-Ansary
Swedish Institute of Computer Science
34 PUBLICATIONS **599** CITATIONS
SEE PROFILE

Seif Haridi
KTH Royal Institute of Technology
227 PUBLICATIONS **4,088** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project M.S. Thesis View project

Project Concurrent Constraint Programming View project

All content following this page was uploaded by Seif Haridi on 21 December 2013.

The user has requested enhancement of the downloaded file.

DTL: Dynamic Transport Library for Peer-To-Peer Applications

Riccardo Reale[1], Roberto Roverso[1,2], Sameh El-Ansary[1], and Seif Haridi[2]

[1] Peerialism Inc, Stockholm, Sweden
{riccardo, roberto, sameh}@peerialism.com
[2] KTH-Royal Institute of Technology, Stockholm, Sweden
{haridi}@kth.se

Abstract. This paper presents the design and implementation of the Dynamic Transport Library (DTL), a UDP-based reliable transport library, initially designed for - but not limited to - peer-to-peer applications. DTL combines many features not simultaneously offered by any other transport library including: *i*) Wide scope of congestion control levels starting from less-than-best-effort to high-priority, *ii*) Prioritization of traffic relative to other non-DTL traffic, *iii*) Prioritization of traffic between DTL connections, *iv*) NAT-friendliness, *v*) Portability, and *vi*) Application level implementation. Moreover, DTL has a novel feature, namely, the ability to change the level of aggressiveness of a certain connection at run-time. All the features of the DTL were validated using a controlled environment as well as the Planet Lab testbed.

1 Introduction

Looking at the rich and diverse requirements of applications in a quickly emerging field like P2P computing, we find that these needs have driven innovation in Internet transport protocols. In the past few years, it has been more and more common that a P2P application develops its own congestion control algorithm, because using out-of-the box TCP congestion control did not suffice. To name a few examples, for a voice and video conferencing application like Skype, a steady low-jitter flow of packets is required. On top of that, due to its real-time nature, Skype traffic must benefit from higher priority with respect to any other application's transfer. Thus, Skype developed an application-level proprietary congestion control algorithm [8] known to be very aggressive towards other applications. On the other end of the spectrum, a content distribution application like Bittorrent, started initially by using TCP but then switched to LEDBAT, in order to be polite as much as possible towards other applications, while saturating the link capacity. Politeness was critical to eliminate the reputation of Bittorrent as a protocol which totally hogs the bandwidth and makes all other applications starve. Between extreme politeness and aggressiveness, many other applications can settle for traditional fair contention over the bandwidth.

The collective state-of-the-art in congestion control algorithms has already addressed most of the diverse needs of P2P applications. The serious shortcoming is the that the best known techniques are scattered around in different libraries. This makes it rather hard for anyone developing a new P2P application to benefit from the progress in the field. This scattering of ideas in addition to some practical and algorithmic issues that we faced while studying other algorithms like LEDBAT, MulTCP and MulTFRC motivated our work on the DTL transport library. The idea is to have one single library that can practically serve as single one-stop-shop for any P2P application.

The following is a list of the target requirements that we realize in DTL:
Priority Levels Supported. The library has to support all levels of traffic prioritization that already exist in the congestion control state-of-the-art.
Inter-Protocol Prioritization. The library has to give control to the application on how polite or aggressive it wants to be against other applications.
Intra-Protocol Prioritization. The library has to give control to the application on how polite or aggressive each individual connection is with respect to other connection within the application. A P2P storage application might need, for instance, to have low-priority background transfers for backup purposes and high-priority transfer to fetch a file that has just been shared by another peer.
In addition to that, there are some features that have to be there for practical purposes:
NAT-Friendliness. The library has to be based on UDP. This makes it easier to circumvent NAT constraints as UDP NAT traversal procedures are known to be more effective than TCP ones [17][11].
Application-level implementation. The library has to be implemented in user space, because for all practical purposes, a kernel-level implementation would hinder any successful wide use. A feature lacking from a rather flexible congestion control library like MulTCP.
Portability. The library shall be available on all platforms, again, that if we want it to be widely-adopted. One final feature which we have not previously found in any other library is the ability not only to specify different priorities for different connection but to be able to change the priority of a particular connection at run-time:
Run-Time Dynamic Prioritization. The need for this feature has arisen, when our team was working on a P2P video streaming application and we needed to tweak the priority of incoming transfers to achieve the necessary playback rate; this by progressively increasing the aggressiveness towards other traffic in the network. An additional advantage of on-the-fly priority tuning is that it prevents disruptions of existing connections and avoids the need of connection re-establishments for transitioning from one priority level to the other or from a congestion control algorithm to the other. In fact, it is known that connection establishments procedures are usually very costly, on the order of several seconds [17], due to expensive peer-to-peer NAT Traversal and authentication procedures.
Finally, the library should support TCP-like reliability and flow control for ease of use.

In this paper, we present how we achieved meeting all aforementioned requirements in the software we call *Dynamic Transport Library* or *DTL*. The paper is organized as follows: in Section 2 we present the state of the art which constitutes the starting point of our effort. With respect to that, we detail the contribution of our work in Section 3. In Section 4 we explain the design and implementation of the library, while in Section 5, we present our library's evaluation results. We then conclude with some final considerations and future work in Section 6.

2 Related Work

LEDBAT is widely accepted as an effective solution to provide a less-than-best-effort data transfer service. Initially implemented in the μTorrent BitTorrent client and now separately under discussion as an IETF draft [1], LEDBAT is a delay-based congestion control mechanism which aims at saturating the bottleneck link while throttling back transfers in the presence of flows created by other applications, such as games or VoIP applications. The yielding procedure is engineered to avoid disruptions to other traffic in the network, and it is based on the assumption that growing one-way delays are symptoms of network congestion. With respect to classical TCP, this allows for earlier congestion detection. LEDBAT was inspired by previous efforts like TCP-Nice [18] and TCP-LP [13] which are based on the same idea.

Our initial goal was to test LEDBAT for background data transfers and later try to tweak its parameters to obtain increased aggressiveness and thus higher transfer priority. Although positively impressed by the ability of LEDBAT to minimize the latency introduced on the network, we found out that tweaking gain and target delay metrics does not lead to a corresponding degree of aggressiveness, as also stated by [5]. As a consequence, we started to investigate other possible solutions for obtaining tunable aggressiveness transfers. MulTCP [9] provides a mechanism for changing the increment and decrement parameters of the normal TCP's AIMD [3] algorithm to emulate the behavior of a fixed number N of TCP flows in a single transfer. The idea of multiple flow virtualization is very promising, however the protocol has been experimented only as a kernel module and it is unclear how an application-level implementation would perform.

MulTFRC [10] extends a previous protocol called TCP Friendly Rate Control (TFRC) [12] to achieve variable transfer aggressiveness. The core idea of TFRC is to achieve TCP-friendliness by explicitly calculating an equation which approximates the steady-state throughput of TCP. MulTFRC modifies the TFRC congestion equation, resulting in a tunable version of the rate control which emulates the behavior of N TFRC flows while maintaining a smooth sending rate. It has been shown that MulTFRC gives better bandwidth utilization than other protocols, such as the Coordination Protocol (CP), by better approximating the steady-state throughput of N virtualized TCP flows [10]. MulTFRC differs from MulTCP in the way that it provides a smoother sending rate, making it partic-

	Inter-protocol prioritization	Intra-protocol prioritization	Application level	Portable	NAT-friendly	Runtime Dynamic Tuning
LEDBAT	L-T-B-E		X		UDP	
MulTCP	L-H, CONT	X			TCP	
MulTFRC	L-H, CONT	X	X		UDP	
DTL	L-T-B-E & L-H, CONT	X	X	X	UDP	X

Table 1: Comparison between protocols. L.T.B.E. = Less-Than-Best-Effort, CONT = Continuous Range, L-H. = Low to High

ularly suitable for multimedia applications. Unfortunately, MulTFRC does not allow for the runtime modification of the N parameter and thus, it is not suitable for our goals [2].

Since MulTCP and MulTFRC are both packet-based congestion control algorithms, if configured to simulate less than one TCP flow, they are likely to be more aggressive than other less-than best effort alternatives. For simulating multiple flows instead, both MulTCP and MulTFRC perform reasonably well with values up to $N = 10$, but only MulTFRC increases linearly for higher values, as mentioned in [10].

3 Contribution

In this paper, we present the design and implementation of an application-level library which provides TCP-like reliability and flow control while implementing variable and configurable on-the-fly traffic prioritization through different congestion control techniques. To the best of our knowledge, this library is first of its kind given the aforementioned characteristics.

The library implements two state of the art congestion controls algorithms: LEDBAT and MulTCP. The two mechanisms have been combined to achieve traffic prioritization which cover a range of levels which start from *less-than-best-effort*, where transfers totally yield to other traffic, up to *high*, where transfers try to reclaim bandwidth from both other intra- and extra-protocol transfers. The priority level can be adjusted using a unique configurable parameter named *priority*. The parameter can be changed at runtime without causing the flow of data to be disrupted and without the need of connection re-establishments. For presentation's purpose, the traffic levels are classified in two operational modes, according to which congestion control algorithm is used:

$$Mode = \begin{cases} Polite, & \textit{if priority} = 0(LEDBAT) \\ Variable, & \textit{if priority} > 0(MULTCP) \end{cases} \tag{1}$$

As a further contribution, the library is implemented in Java as an attempt to make the software portable between different platforms. We are unaware of any other effort to fully implement LEDBAT, MulTCP or any relevant congestion control mechanism for that matter on an application-level library in Java. A feature-wise comparison of the state-of-the art against DTL is shown in Table 1.

4 Dynamic Transport Library (DTL)

We implemented *DTL*, using Java NIO over UDP. We drew inspiration for its design from TCP with SACK (Selective Acknowledgment Options). For this reason, we provide an application level TCP-like header in order to enable reliability, congestion control and flow control over UDP in the same way TCP does. In our implementation, the header is appended to each datagram together with the data to be transmitted, and it is encoded/decoded by the sender and receiver modules at the respective ends of the connection. The base header carries the receiver's advertised window, the sequence number and the acknowledge number. The packet size is dynamically chosen. By default, *DTL* uses large packet sizes of 1500 bytes, while at slow rates the size can be adjusted down to 300 bytes.

The sender's module which initiates a transfer maintains three variables: (1) the congestion window *cwnd*, meant as a sender-side limit, (2) the receiver's advertised window, as receiver-side limit, and(3) the slow start threshold.

End-to-end flow control, used to avoid the sender transmitting data too quickly to a possible slow receiver, is obtained using a sliding window buffer. In every acknowledgement packet, the receiver advertises its receive window as the amount of data that it is able to buffer for the current connection. At each point in time, the sender is allowed to transmit only a number of bytes defined as follows:

$$allowed = min(rcv\_win, cwnd) - inflight \tag{2}$$

where rcv_win is the advertised receive window, $inflight$ is the amount of data in transit on the network and not yet acknowledged, while *cwnd* is the congestion window size.

Reliability is ensured by tagging each packet with a sequence number which corresponds to the the amount of bytes sent up to that point in time. Using the sequence number, the receiver is able to understand the ordering of packets and identify losses. For each received packet, the receiver sends back an acknowledgement(*ack*) with the amount of bytes it has received up till that point. The sender can detect packet loss in two ways: when a packet times out, or when the receiver notifies the sender with a special format's selective acknowledgement. The packet timeout value is correlated with the estimated RTT. The latter is updated the same way TCP does: using the Karn/Partridge algorithm [15].

The Selective Acknowledgement is generated by the receiver after more than three out-of-order packets, and contains the information about all successfully received segments. Consequently, the sender can retransmit only the segments which have actually been lost. The library's behaviour in case of packet loss or *ack* reception is defined at the server-side by the chosen congestion control mechanism.

This is because, while in our library flow control and reliability designs are both directly derived from TCP, we provided a different implementation of congestion control according to the transfer prioritization policy which needs to be enforced.

In general terms, in order to control the aggressiveness of the flow, we provide the applications with *priority* parameter for each socket, intended as a positive floating point number. The correlation between the priority parameter and the service mode has been previously defined in Equation 1.

In the *polite* mode, we make use of the LEDBAT algorithm. In the *variable* mode instead, we provide our implementation of MulTCP, parametrized with a priority value corresponding to the number $N == priority$ of flows to virtualize. The choice of MulTCP is motivated by the fact that, while MulTFRC might provide a small improvement in transfer stability and bandwidth utilization [2], the number of virtualized flows cannot be changed at runtime. In the following two sections, we will detail our implementation of both *polite* and *variable* modes, describing the implementation of the congestion control algorithms.

4.1 Polite Mode

Although LEDBAT is not the only congestion control providing less-than-best-effort transfers, it is the only one which actually tries to control the latency introduced on the network by the transfer itself. Other similar algorithms, like TCP-Nice [18] and TCP-LP [13] use the increasing delay as an indicator of imminent congestion as LEDBAT does, but they try to react in a more conservative manner, or simply by backing off earlier than TCP. LEDBAT instead opts to keep the delay under a certain threshold, reason for which it is able to achieve a yielding factor that is higher than other congestion controls, as explained in [5].

Since the main congestion indicator in LEDBAT is the one-way delay variation, in order to react earlier than TCP to congestion events, we added a timestamp and delay header fields to compute its value. For every packet sent, the sender appends to the packet its own timestamp, while the receiver sends back an acknowledgement containing that same timestamp (used also for better RTT estimation) and the measured one-way delay. The sender's congestion control module maintains a list of the minimum one-way delays observed every minute in a $BASE\_HISTORY$ queue. The smallest delay D_{min} is used to infer the amount of delay due to queuing, as we assume it represents the physical delay of the connection before any congestion happened. The mechanism of using only "recent" measurements, letting the old ones expire, results in a faster response to changes in the base delay. Consequently it also allows to correct possible errors in the measurements caused by clock skewness between sender and receiver. The last measured one-way delays are stored in a $NOISE\_FILTER$ queue. The lowest value in the queue is used to compute the queuing delay. In our implementation, the $BASE\_HISTORY$ memory time is set to 13 minutes while the $NOISE\_FILTER$ queue contains the 3 last measured delays.

The key element in the LEDBAT congestion control algorithm lies in comparing the estimated queuing delay against a fixed target delay value τ, considered as the maximum amount of delay that a flow is allowed to introduce in the queue of the bottleneck buffer. The difference $\Delta(t)$ between the queuing delay and the target is used to proportionally increase or decrease the congestion window. The original LEDBAT linear controller is defined as follows:

$$\Delta(t) = \tau - (D(t) - D_{min}) \tag{3}$$

$$cwnd(t+1) = cwnd(t) + \gamma\Delta(t)/cwnd(t) \tag{4}$$

where γ is the gain factor. The controller has a behavior similar to TCP in the way it reacts to a packet loss by halving the congestion window.

In our implementation, we set $\gamma = 1/TARGET$, so that the max ramp-up rate is the same of TCP, and $\tau = 100ms$, as specified in the BitTorrent's open source μTP implementation and later confirmed in the second version of the LEDBAT Internet draft (July 2010).

A first implementation of the standard LEDBAT linear controller confirmed the presence of intra-protocol fairness issues, known as the late-comer advantage. Primary causes of the problem are *base delay measurement errors* and a wrong *window decrement policy.* In order to overcome this issue we implemented the solution proposed by Rossi et al. [16], i.e. applying the TCPs slow-start mechanism to the very beginning of LEDBAT flows. Slow-start forces a loss in the other connections active on the same bottleneck, thus allowing the new-coming flow to measure a correct base delay.

From our experiments however, we found out that even using the slow-start mechanism, slight differences in the base delay measurements might lead to significant unfairness among transfers. We identified the problem to be in the *Additive Increase/Additive Decrease (AIAD)* mechanism. Referring to the work of Chiu et Al.[7], we implemented a a simple Additive Increase/Multiplicative Decrease (AIMD) algorithm instead, which is proven to guarantee stability. As a result, we modified the original Equation 4 in such a way that, if the estimated queuing delay exceeds the *tau* value, the *cwnd* shrinks by a $\beta < 1$ factor, as described here:

$$cwnd(t+1) = \begin{cases} cwnd(t) + \gamma\Delta(t)/cwnd(t) & \mathit{if}\ \Delta(t) >= 0 \\ cwnd(t) \times \beta & \mathit{if}\ \Delta(t) < 0 \end{cases} \tag{5}$$

With this modification, the decrement of the congestion window is caused by the queuing delay reaching a higher value than the target. The decrement also becomes *proportional* to the sending rate itself. Flows with higher sending rate will then decrease more than others. In our implementation, we used a β factor of 0.99 as found in other implementations [6].

The validity of our considerations has been confirmed by an independent study [6], in which the authors analytically prove that the *additive-decrease* component in the LEDBAT linear controller makes the system unfair, causing transfers to fall out of Lyapunov stability. In the same study, two solutions are proposed for this problem: the first, more conservative, consists of adding a probabilistic drop to the additive increase/decrease dynamics. The other, more aggressive, directly replaces the additive decrease with a multiplicative one, thus confirming our finding.

4.2 Variable Mode

We implemented our variable mode transfer priority using the MulTCP algorithm. MulTCP [9] congestion control mechanism allows for a single flow to behave like an aggregate of N concurrent TCP connections, in particular from the point of view of the throughput.

The congestion control module is implemented on top of the normal TCP SACK algorithm using the *priority* value as number of virtual flows N which one transfer must virtualize. MulTCP simply provides the a and b parameters in the additive-increase/multiplicative-decrease (AIMD) algorithm which are proportional to the number of flows to emulate. MulTCP tries to closely emulate the behavior of TCP in all of its phases, so that in both slow start and congestion avoidance, the congestion window grows as N TCP flows would. In order to avoid sending large bursts of packets if N is too large, causing packet losses, we also implemented the smooth slow start algorithm introduced in [9].

As a further improvement to the original MulTCP algorithm, we implemented the *MulTCP2* algorithm modification proposed by Nabeshima et al. [14]. The mechanism makes it possible, in case of packet loss, to achieve a better virtualized behavior than the original MutlTCP specification, considered to be too aggressive. Here we report the suggested equation implemented in our library.

First, the steady-state average congestion window of N flows is calculated as follows:

$$cwnd_N = N \times cwnd_{std} = \frac{N\sqrt{1.5}}{\sqrt{p}} \tag{6}$$

where p is the packet loss rate. Then we derive the $cwnd$ in function of a, AIMD's increase parameter, and b, AIMD decrease parameter:

$$cwnd = \frac{\sqrt{a(2-b)}}{\sqrt{2bp}} \tag{7}$$

Finally b is derived from equations 6 and 7 as:

$$b = \frac{2a}{a + 3N^2} \tag{8}$$

We then set a to be N, the number of virtualized flows, which is also equal to our *priority* parameter, and $b = 2/(1 + 3N)$.

It's easy to observe that, for $a = 1$, the parameters of the AIMD algorithm are exactly the same of the standard TCP.

5 Evaluation

In this chapter, we report the results gathered using DTL. Our evaluation methodology is the same used for studies conducted on the LEDBAT protocol such as [16].

We performed our tests in two different configurations:

– a controlled network environment consisting of three host machines using Ubuntu with kernel 2.6.32-22, two as traffic sources and one as traffic sink, connected by a Gigabit links. In order to emulate bottleneck network conditions, we used the *Dummynet* traffic shaper [4]. We created two pipes with RED queue and standard periodic loss, the first with symmetric capacity of $C = 1Mbps$ for the low-bandwidth scenario, and one with symmetric capacity of $C = 10Mbps$, for the high-bandwidth scenario. Both of them are configured with a delay of $20ms$. This is a common configuration setup used by other related works [10][9].
– a real-world environment using the PlanetLab testbed. The experiments are performed using the same configuration of the controlled case, i.e. two hosts, in this case PlanetLab machines, as traffic sources and one as traffic sink. Only non-congested nodes with a symmetric capacity of $C = 10Mbps$ are chosen to host the experiment.

As metrics to evaluate the performance of the DTL, we adopted:

1. the notion of *fairness* introduced by Jain's fairness index F [7], defined as:

$$F = \frac{(\sum_{i=1}^{N} X_i)^2}{N \cdot \sum_{i=1}^{N} X_i^2} \tag{9}$$

where x_i is the rate of flow i and N is the number of flows sharing the bottleneck. Notice that, when N flows get the same bottleneck share, fairness is equal to 1, while it decreases to $1/N$ in the unfair case, where a single flow overtakes the other transfers and uses all available capacity.
2. the *efficiency* or link utilization η, defined as the ratio of the total link utilization normalized over the available bandwidth.
3. the *normalized throughput*, when comparing the total throughput of multiple parallel flows.

We identify flows with dynamic priority over time as DTL(0,1,2), where 0,1 and 2 are the priority values at different point in time in the experiment.

5.1 Polite Mode results

Early results of our initial implementation of LEDBAT, following the draft specification, with the slow start option enabled, immediately showed three fundamental characteristics:

– Capacity of the congestion control algorithm of exploiting all the available bottleneck resources, keeping the one-way delay around the queuing delay target and yielding to TCP flows.
– Fairness in sharing available bandwidth in case of multiple flows starting at the same point in time.
– Unfairness in sharing available bandwidth when flows start at different point in time, even when using the slow start mechanism.

Figure 1 shows a comparison of two intra-protocol examples. The first run (a) is executed using the original algorithm with slow-start, while the second (b) using the modified algorithm with multiplicative-decrease. The results have been obtained using the first test configuration. As shown, the original linear controller is not able to timely compensate the initial small error in the base-delay measurements, while the introduction of the non-linear decrease seems to efficiently solve the issue. We would like to underline that, from our experiments, both techniques, i.e. slow start and multiplicative decrease, should be used to guarantee intra-protocol fairness as shown on Figure 1(b). As opposed to another previous which claims that implementing multiplicative-decrease is enough to provide such behaviour.

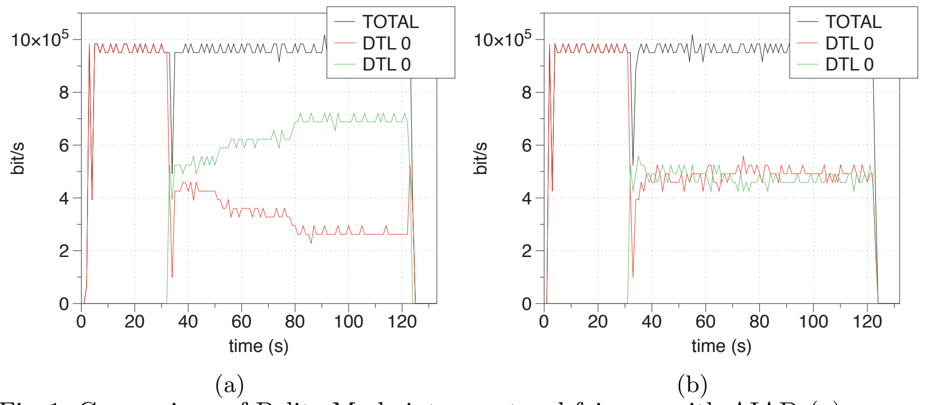


Fig. 1: Comparison of Polite Mode intra-protocol fairness with AIAD (a) as opposed to with AIMD (b)

In Figure 2 we plotted the temporal evolution of the throughput of ten DTL flows with $priority = 0$ (DTL0), starting 5 seconds after each others with a duration of 300 seconds. The test was executed on PlanetLab. The low priority flows perfectly share the bottleneck using all the available resources.

We present in Figure 4 the temporal evolution of two polite flows (DTL0) and two TCP flows on the PlanetLab configuration. The two DTL0 flows, one started at $t = 0$ and the other at $t = 60$, equally share the available bandwidth, then yield to a TCP flow at $t = 120$, and again fill up the bottleneck link when the TCP flows, the second one having joined at $t = 180$, terminate at $t = 240$.

Finally, we examine the impact of the Multiplicative-decrease modification in terms of efficiency η, fairness F and loss rate L. We considered two low-priority flows starting at different points in time. Both flows share a common bottleneck both in low and high bandwidth configuration. Each simulation lasted 300 seconds. In Table 2, we report the average of multiple simulation runs executed at different point in time. The results refer to the time interval where both flows are

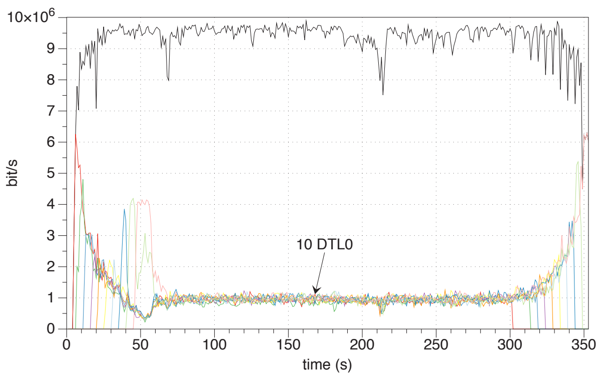


Fig. 2: Temporal evolution of the throughput of 10 DTL 0 flows on the path planetlab1.sics.se - planet2.zib.de

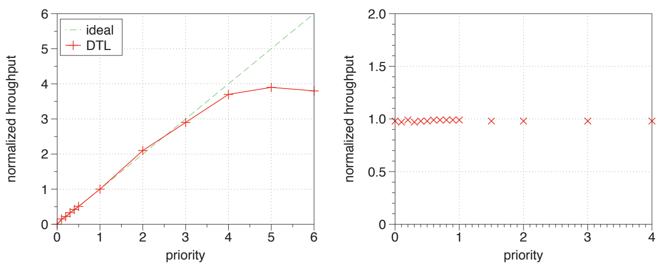
active at the same time. The gathered results clearly demonstrate the positive effect of the multiplicative-decrease modification on the intra-protocol fairness in both bottleneck configurations. As shown, the modification does not affect bandwidth usage and it causes significantly less packet loss.

	C	η	F		L	
	$Mbit$	[%]	μ	σ	μ	σ
No multiplicative decrease	1	99	$5.9 \cdot 10^{-1}$	$1.01 \cdot 10^{-1}$	$1.8 \cdot 10^{-2}$	$5.03 \cdot 10^{-3}$
	10	98	$6.68 \cdot 10^{-1}$	$1.87 \cdot 10^{-1}$	$1.68 \cdot 10^{-2}$	$6.20 \cdot 10^{-3}$
With multiplicative decrease	1	98	$9.88 \cdot 10^{-1}$	$1.03 \cdot 10^{-2}$	$5.4 \cdot 10^{-3}$	$1.2 \cdot 10^{-4}$
	10	98	$9.92 \cdot 10^{-1}$	$4.90 \cdot 10^{-3}$	$8.08 \cdot 10^{-3}$	$7.60 \cdot 10^{-5}$

Table 2: Polite mode $priority = 0$ (fixed): results showing the Link utilization average η, together with the observed Fairness F and packet Loss Rate L, both considering their mean μ and standard deviation σ values

5.2 Variable Mode results

For *priority* parameter values greater than zero, the congestion control switches to the MulTCP algorithm, which uses packet loss as congestion detection rather than increased delay as in LEDBAT. In Figure 3(a), we present the normalized throughput value of one DTL flow parametrized with different priorities, ranging from 1 to 6, against nine TCP flows. In the experiment, all ten flows share a bottleneck link of capacity $C = 10Mbits$. As shown, the DTL flow reaches a throughput value of about *priority* times the one of a single TCP flow up to *priority* value of 4, which leads to the best trade-off in terms of throughput. This effectively means that, for $priority = 4$, the DTL flow appears as 4 TCP flows combined, leaving to each of the other nine TCP flows a share of 1/13



(a) (b)

Fig. 3: Plot of the relative Throughput (a) and the Intra-protocol Fairness (b) as a function of *priority*

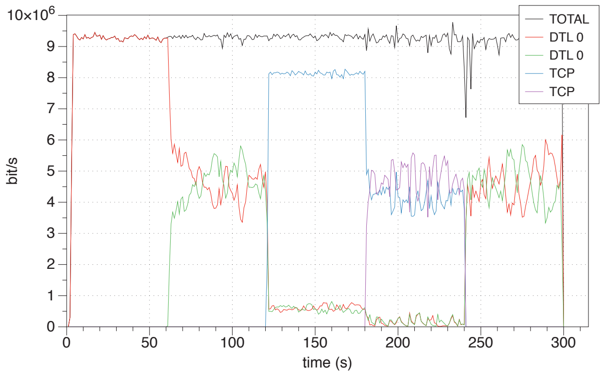


Fig. 4: Temporal evolution of the throughput of two DTL flows $priority = 0$ (fixed) and two TCP flows on the Planetlab testing configuration

of the bandwidth. However, for values of *priority* greater than 4, the link gets congested and no more improvements are possible due to the high rate of packet loss. In Figure 3(b) instead, we compare the normalized throughput of one DTL flow with respect to another DTL flow of same *priority*, for values of *priority* ranging from 1 to 4. This in order to show how the intra-protocol fairness is maintained for all priorities, as the normalized throughput remains 1.0 for all *priority* parameters. Figure 5 presents the same PlanetLab test scenario as in Figure 4, but this time setting $priority = 1$. As expected, the DTL flows share in a fair manner the bandwidth resources with the two new-coming TCP flows.

Similarly to the polite mode, we decided to examine with more accuracy the DTL behavior for $priority = 1$ in terms of efficiency η, fairness F and $Normalized\ Throughput$. We considered two DTL1 flows for the intra-protocol configuration, and a single DTL1 flow against a TCP flow for the inter-protocol one. Each simulation has a duration of 300 seconds. In Table 3, we report the average and standard deviation of multiple runs. The results confirm the ability of DTL1 to correctly share the bandwidth capacity with other TCP flows competing on the same bottleneck under the same conditions, MTU- and RTT-wise.

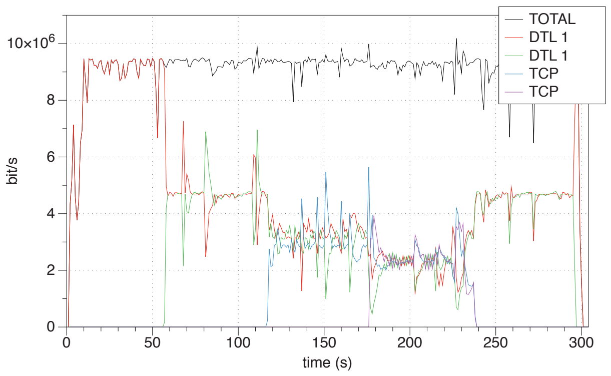


Fig. 5: Temporal evolution of the throughput of two DTL flows $priority = 1$ (fixed) and two TCP flows on the Planetlab testing configuration

	C $Mbit$	η [%]	F μ	F σ	$Normalized\ Throughput$ μ	$Normalized\ Throughput$ σ
Intra-protocol	1	97	$9.93 \cdot 10^{-1}$	$3.7 \cdot 10^{-3}$	$9.92 \cdot 10^{-1}$	$8.56 \cdot 10^{-2}$
	10	98	$9.96 \cdot 10^{-1}$	$9.3 \cdot 10^{-4}$	$9.84 \cdot 10^{-1}$	$2.3 \cdot 10^{-2}$
Inter-protocol	1	97	$9.83 \cdot 10^{-1}$	$3.5 \cdot 10^{-3}$	$9.84 \cdot 10^{-1}$	$4.47 \cdot 10^{-2}$
	10	98	$9.52 \cdot 10^{-1}$	$8.6 \cdot 10^{-3}$	$9.85 \cdot 10^{-1}$	$2.15 \cdot 10^{-2}$

Table 3: Variable mode $priority = 1$(fixed): results showing the Link utilization average η, together with the observed Fairness F and $Normalized\ Throughput$

We then present an experiment in Figure 6, using the first test configuration, where DTL flow's priority value changes with time. The simulation lasts 480 seconds, the bottleneck is set to 5Mbit with RTT = 25ms. In order to emphasize the different degrees of aggressiveness, we introduce a TCP transfer on the same bottleneck. The DTL flow starts in polite mode and completely yields to the TCP flow until second 240 when its priority is updated to $priority = 1$, forcing it to switch from LEDBAT to MulTCP. Just after that point, the DTL flow starts to increase its congestion window and share in a fair manner the bandwidth with TCP. Finally, at second 360, we increase the priority up to 2. The congestion window grows now twice as fast. However, in case of packet loss, the same windows is decreased by a value of less than a half, depending on the value of b. The result is a throughput twice as large as in TCP. In Figure 7, we show a plot, produced in our first configuration, of five DTL flows with $priority$ value which varies in time. At first, all flows start with $priority = 0$, then at second 120 four of them get updated to $priority = 1$. We can clearly observe that the four DTL1 flows share the bandwidth equally, while the only DTL0 flow yields to them. At second 230 two of the flows get upgraded to $priority = 2$ and the others keep their previous priority value. As a result, the DTL2 flows consume equally more bandwidth than the other two DTL1 flows, while the DTL0 still yields to all of them. These results confirm our expectations of multiple priorities intra-protocol fairness.

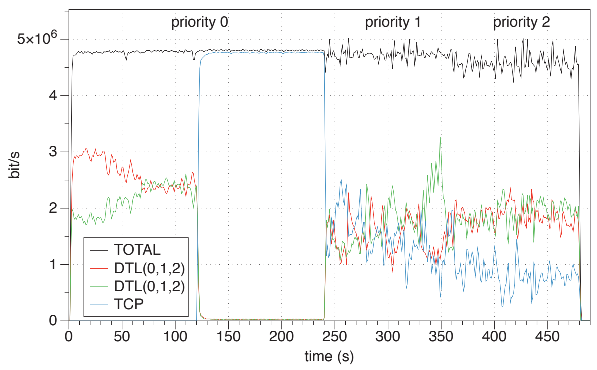


Fig. 6: Temporal evolution of the throughput of two dynamic priority DTL(0,1,2) flows which share the bottleneck link with a TCP flow

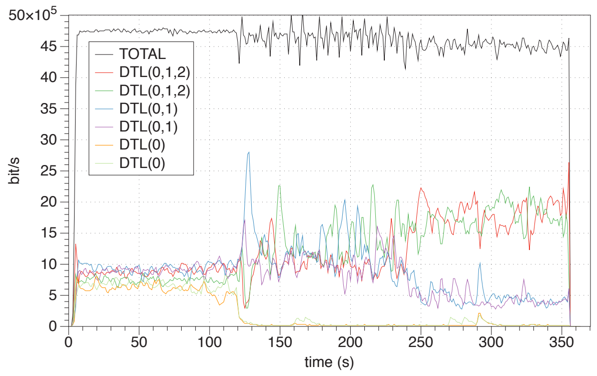


Fig. 7: Temporal evolution of the throughput of six DTL flows with varying *priority* value

6 Conclusion

In this paper we presented the design and implementation of the DTL application-level library, which is a reliable, variable priority transfer library developed in the Java language, using NIO over UDP. In order to provide different transfer prioritization policies, we implemented two state-of-the art congestion control control mechanisms: LEDBAT and MulTCP. We motivated our choices in using the aforementioned algorithms for the case of configurable priority. As an important achievement, we showed in our results that the library performs on-the-fly transfer priority changes as required, while avoiding connection termination or transfer rate fluctuations. On top of that, our results obtained both using a controlled environment and the Planet Lab testbed show that the library meets all necessary fairness and throughput requirements under the implemented priority levels.

As future work, we would like to provide a more extensive evaluation of our library in a deployed peer-to-peer system. We also would like to investigate the possibility of modifying the MulTFRC algorithm, which provides a more stable approximation of multiple TCP flows behavior than MultTCP, to support variable priority. We plan to make DTL available as an open source project in the near future.

References

1. Ledbat ietf draft. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-03 (July 2010)
2. Multfrc ietf draft. http://tools.ietf.org/html/draft-irtf-iccrg-multfrc-01 (July 2010)
3. Allman, M., Paxson, V., Blanton, E.: Tcp congestion control (9 2009), http://www.ietf.org/rfc/rfc5681.txt
4. Carbone, M., Rizzo, L.: Dummynet revisited. Computer Communication Review 40(2), 12–20 (2010)
5. Carofiglio, G., Muscariello, L., Rossi, D., Testa, C.: A hands-on assessment of transport protocols with lower than best effort priority. CoRR abs/1006.3017 (2010)
6. Carofiglio, G., Muscariello, L., Rossi, D., Valenti, S.: The quest for ledbat fairness. CoRR abs/1006.3018 (2010)
7. Chiu, D.M., Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Computer Networks 17, 1–14 (1989)
8. Cicco, L., Mascolo, S., Palmisano, V.: An experimental investigation of the congestion control used by skype voip. In: Proceedings of the 5th international conference on Wired/Wireless Internet Communications. pp. 153–164. WWIC '07, Springer-Verlag, Berlin, Heidelberg (2007)
9. Crowcroft, J., Oechslin, P.: Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. CoRR cs.NI/9808004 (1998)
10. Damjanovic, D., Welzl, M.: Multfrc: providing weighted fairness for multimediaapplications (and others too!). Computer Communication Review 39(3), 5–12 (2009)
11. Guha, S., Francis, P.: Characterization and measurement of tcp traversal through nats and firewalls. In: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement. pp. 18–18. IMC '05, USENIX Association, Berkeley, CA, USA (2005), http://portal.acm.org/citation.cfm?id=1251086.1251104
12. Handley, M., Floyd, S., Padhye, J., Widmer, J.: Tcp friendly rate control (tfrc): Protocol specification (2003)
13. Kuzmanovic, A., Knightly, E.W.: Tcp-lp: low-priority service via end-point congestion control. IEEE/ACM Trans. Netw. 14(4), 739–752 (2006)
14. Nabeshima, M.: Performance evaluation of multcp in high-speed wide area networks. IEICE Transactions 88-B(1), 392–396 (2005)
15. Paxson, V., Allman, M.: Computing tcp's retransmission timer (2000)
16. Rossi, D., Testa, C., Valenti, S., Veglia, P., Muscariello, L.: News from the internet congestion control world. CoRR abs/0908.0812 (2009)
17. Roverso, R., El-Ansary, S., Haridi, S.: Natcracker: Nat combinations matter. In: Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks. pp. 1–7. ICCCN '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICCCN.2009.5235278
18. Venkataramani, A., Kokku, R., Dahlin, M.: Tcp nice: A mechanism for background transfers. In: OSDI (2002)

View publication stats