Security Analysis of the Micro Transport Protocol with a Misbehaving Receiver

Florian Adamsky*, Syed Ali Khayam†, Rudolf Jäger‡, Muttukrishnan Rajarajan*

*City University London, United Kingdom
Email: {Florian.Adamsky.1, R.Muttukrishnan}@city.ac.uk

†National University of Sciences and Technology, Pakistan
Email: Ali.Khayam@seecs.nust.edu.pk

‡Technische Hochschule Mittelhessen University of Applied Sciences, Germany
Email: Rudolf.Jaeger@iem.th-mittelhessen.de

***Abstract*—BitTorrent is the most widely used Peer-to-Peer (P2P) protocol and it comprises the largest share of traffic in Europe. To make BitTorrent more Internet Service Provider (ISP) friendly, BitTorrent Inc. invented the Micro Transport Protocol (uTP). It is based on UDP with a novel congestion control called Low Extra Delay Background Transport (LEDBAT). This protocol assumes that the receiver always gives correct feedback, since otherwise this deteriorates throughput or yields to corrupted data. We show through experimental investigation that a misbehaving uTP receiver, which is not interested in data integrity, can increase the bandwidth of the sender by up to five times. This can cause a congestion collapse and steal large share of a victim's bandwidth. We present three attacks, which increase the bandwidth usage significantly. We have tested these attacks in a real world environment and show its severity both in terms of number of packets and total traffic generated. We also present a countermeasure for protecting against the attacks and evaluate the performance of that defence strategy.**

I. Introduction

With the widespread use of handheld devices the Internet traffic is increasing at an enormous rate. This traffic can be classified in two categories: background and foreground traffic. Background traffic is e.g. operating system (OS) updates, P2P application or backup transfer and does always have a lower priority then foreground traffic. In contrast, foreground traffic is e.g. Email, Voice over IP (VoIP) or web browsing and an extra delay is not acceptable. To separate these traffic categories it is necessary to define static firewall rules. This approach is inflexible and often leaves big head room which is unused. A better approach is to use a separate transport protocol for background traffic like TCP-LP [1] or TCP-Nice [2]. These protocols can detect foreground traffic and automatically reduce their sending rate. With uTP using LEDBAT there is a new kid on the block.

In December 2008, BitTorrent announced in the developer forum that μTorrent will switch the data transfer from TCP to UDP [3]. Shortly after that announcement, panic started spreading all over the media. The media named it "*The Next Internet Meltdown*" [4], because of the missing congestion avoidance algorithms in UDP. However, they all got it wrong. BitTorrent clarifies that the new uTP will have a novel congestion avoidance algorithm [5]. After that statement the technology information website *Ars Technica* published an article with the title "μTorrent switch to UDP and why the sky isn't falling" [6].

To the best of our knowledge, this research work is the first security analysis of the uTP using the LEDBAT congestion control. Since October 2009 the congestion control LEDBAT has a IETF draft [7]. The security considerations from this draft only describes a network, where a malicious node in between, delays packets unnecessarily to lower traffic throughput.

II. Background

This section provides an overview of how uTP works. To better understand why BitTorrent switched to UDP, let us recall what the problems were with TCP. BitTorrent is usually background traffic. But due to the

978-0-7695-4810-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CyberC.2012.31

fact that it uses multiple TCP connections at once, it gets an unfair advantage over other applications. This can be explained by the fact that TCP distributes the available bandwidth evenly across all connections. A BitTorrent user has to restrict the bandwidth by setting a down- and upload limit in his client, to prevent the client to consume all the available bandwidth. This requires knowledge about its own Internet connection and often leaves big head room which is unused. When there is a need for interactive traffic like VoIP or browsing the web, it is necessary to adjust the down- and upload limits or pause the complete BitTorrent client. These are the reasons why BitTorrent invented the new uTP, which detects unused head room automatically and adjust its limits.

The uTP [8] is on top of UDP. Since UDP has no congestion control, uTP can implement its own one. Like TCP, uTP controls the flow with a sliding window, verifies data integrity with sequence numbers and initiates a connection with a handshake. Unlike TCP, sequence numbers refer to packets instead of bytes and uTP initiates a connection with a two-way handshake, instead of a three-way. uTP also supports Selective Acknowledgment (SACK) via an extension, which is enabled per default. In contrast to TCP, uTP SACK uses a bitmask where each bit represents a packet in the send window, instead of defining ranges. The relation between a bit and the packet is the acknowledgment number. The first bit in the bitmask represents the `ack_nr + 2`, since the `ack_nr + 1` packet is assumed to be lost. If a bit is set, this means we got that packet and vice versa. A set bit for a packet which has not been send yet, will be ignored by the sender. The major difference between uTP and TCP is novel congestion control LEDBAT.

A. Low Extra Delay Background Transport (LEDBAT)

uTP presents a novel congestion avoidance algorithm: LEDBAT. uTP together with LEDBAT tries to solve the above mentioned problems from TCP by using a modem queue size as a controller. If the queue size of the send buffer is too large, then the sender throttles back. To archive that, LEDBAT [7] uses one way delay measurement as the main congestion algorithm, which we will describe in more detail in Section III-B1. Additional to that value, LEDBAT also uses packet loss and the number of acknowledgments during one window. These measurements make it possible to detect foreground traffic (e.g. VoIP, HTTP, . . .). If these measurements differ to much from previous values, the sender adjusts its sending rate accordingly. This automatic adjustment gives foreground traffic a higher priority than BitTorrent with uTP and makes manual adjustments unnecessary. However, this only works if both, the sender and the receiver, cooperates with each other.

III. Attack Analysis

Protocols assumes that the receiver always gives correct feedback. Normally this is correct, since otherwise this deteriorates the throughput or yields to corrupted data [9]. An attacker who is not interested in data integrity can give incorrect feedback, to induce the sender to send more and more packets into the network. The next section will describe possible attack scenarios.

A. Attack Scenarios

Regarding of uTP a misbehaving receiver can create the following attack scenarios:

1) Congestion Collapse: If a high bandwidth victim has a node in between which does not have the same bandwidth capacity, then there is a possibility of congestion. The effects of congestions are: packet loss, queuing delay and block of new connection. Normally the congestion control from the transport protocol takes care of that. However, if a malicious peer can trick the congestion control, then it is possible to create congestion on that path. The consequence of congestion is, that other clients have problems to reach the high bandwidth victim which yields to a denial-of-service (DoS) attack. However, it is not only necessary to have a node in between with lower bandwidth capabilities, a Digitial Subscriber Line (DSL) and cable modem is sometimes enough. Normally DSL and cable modems have a send buffer which is disproportional to their maximum send rate [8]. If a misbehaving receiver induce the sender to fill the buffer of its DSL or cable modem with packets, the sender cannot deliver packets to other peers. If there is no DSL/cable modem or a slow node in between, it is still possible to steal bandwidth from that peer.

2) Steal Bandwidth: Since bandwidth is a limited resource it is important to share that resource fairly. If a malicious peer trick the congestion control to get more bandwidth, other peers will suffer. We have

shown this suffering in our previous research work, where a malicious peer exploit the choking mechanism to get more bandwidth [10]. This attack destabilizes BitTorrent's clustering behavior [11], which attacks the high bandwidth leechers in a swarm. In the next section we will discuss attacks which can trick the congestion control of uTP.

B. Details about the Attacks

For this evaluation we used the open-source library libUTP [1]. This library is written by the developer of BitTorrent and it is used by the following BitTorrent clients: μTorrent, Vuze, Mainline and Transmission. According to the latest client statistics from the P2P research team at the Delft University of Technology, these clients are comprising a market share of 90.47 % [12]. LibUTP comes with test program for receiving (`utp_recv`) and sending files (`utp_send`). While writing this paper, this is the latest version [2] of libUTP.

For the evaluation we used a real world client-server environment. One computer was the sender which had an unmodified version of libUTP. The other computer was the receiver and had a modified version of libUTP. Both computers were running GNU/Linux and were connected via a 100 Mbit/s switch. We introduced 25 ms delay with 10 ms variance which is distributed normally with NetEm [13] on the sender side. We choose these values to loosely simulate a connection where two peers with high speed Internet access (Asymmetric Digitial Subscriber Line (ADSL), ADSL2+) are communicating with each other. For all the tests we used `utp_recv` and `utp_send` to initiate a simple file transfer with a 100 MiB file. This test environment was used for the rest of the results presented in this paper

1) Attack 1: Lying about the Delay: According to the Internet-Draft from the IETF the one way delay measurement is only possible with the help of the receiver: "*LEDBAT requires that each data segment carries a "timestamp" from the sender, based on which the receiver computes the one-way delay from the sender, and sends this computed value back to the sender*" [7]. This one way delay, which we denote as Δt, is calculated on the receiver side by subtracting the timestamp t_{snd} from the current time of the receiver t_{rcv}. The receiver sends Δt back to the sender, as shown in Figure 1(a).

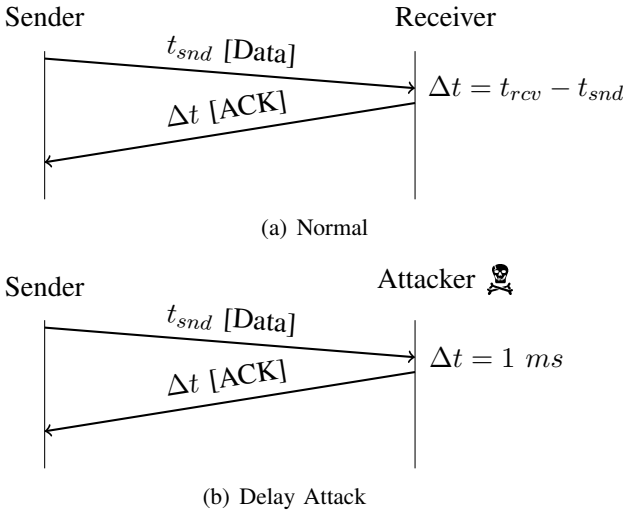


Figure 1. One way delay measurement with a normal receiver and an attacker

The value of Δt is only meaningful as a relative value compared to previous values, because the clocks of the peers are not synchronized. The sender saves all Δt values in a vector $\vec{our\_hist}$ from the last two minutes. Before these values will be included into $\vec{our\_hist}$, Δt will be normalized with *delay_base*, which is the lowest value from $\vec{our\_hist}$. Normalization is done to get a measurement of the current buffering delay on the socket [8]. LEDBAT uses equation 1 to decide whether to increase or decrease the send window. LEDBAT increases the send window, when the lowest value in vector $\vec{our\_hist}$ is smaller then 100 ms. Otherwise the send window will remain constant or gets decreased.

$$\text{off\_target} = \begin{cases} + & \text{if } min(\vec{our\_hist}) < 100\ ms \\ = & \text{if } min(\vec{our\_hist}) = 100\ ms \\ - & \text{if } min(\vec{our\_hist}) > 100\ ms \end{cases} \quad (1)$$

As an attacker it is possible to lie about the delay measurement. Figure 1(b) shows that an attacker pretends that the one way delay is always 1 ms. It is not necessary to run this attack with $\Delta t = 1$ ms, because Δt is not interpreted as an absolute value. It is only essential that Δt is constant and under 100 ms. This yields to the following situation: Shortly after the

[1] https://github.com/bittorrent/libutp

[2] Commit ID: `2c678a26e5`

attack started, *delay_base* will be 1 ms, since this is the lowest value during the last two minutes. All following Δt values will be normalized with *delay_base*. Therefore we can fill the vector $our\_\vec{hist}$ all with zeros. The `delay_factor` makes use of the vector $our\_\vec{hist}$, which is partly responsible for the adjustment of the maximal window. Equation 2 shows how the variable `delay_factor` gets calculated.

$$delay\_factor = \frac{100\ ms - min(our\_\vec{hist})}{100\ ms} \quad (2)$$

Since all values in $our\_\vec{hist}$ are zero, this means that `delay_factor` is always one, which is the highest value. A positive `delay_factor` always increases the window from the sender. The sender receives Δt and increases or decreases its window size `max_window` according to the pseudo code in Listing 1.

Listing 1. Maximal window calculation according to [8]

```
scaled_gain =
    MAX_CWND_INCREASE_PACKETS_PER_RTT
    * delay_factor * window_factor;

max_window += scaled_gain;
```

Figure 2 shows packets per second depending on time in seconds. All curves show the bandwidth usage of a file transfer over time. A file transfer with an unmodified receiver has an average value of 632.503 packets/sec. A modified receiver which lies about the one way delay measurement has an average value of 916.507 packets/sec. This shows that the described attack can increase the bandwidth up to around 300 packets/sec. This attack is limited, since its increase in bandwidth yields to an increase in packet loss. The average number of packets that a sender with a normal behaving receiver assumes to be lost is up to 16761.9. A sender who communicates with a misbehaving receiver the average number of packet loss is up to 19031.3. This is an increase of 13.54 %. Every packet loss decreases the send window of the sender. This leads to the next possible attack.

2) Attack 2: Lazy Optimistic Acknowledgment: Like TCP, uTP also uses packet loss as a sign of congestion and decreases its sending rate. In TCP, when a packet is lost, the sending rate will be multiplied by a factor of 0.5. Since this is a much less likely event in uTP, it

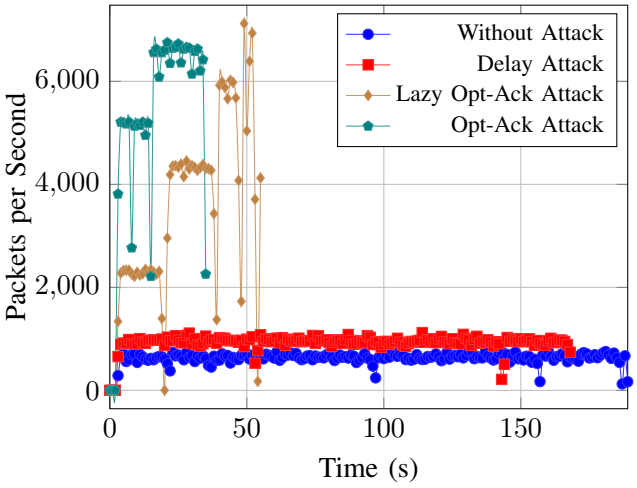


Figure 2. File transfer of a 100 MiB file via uTP under the following network conditions: Bandwidth: 100 Mbit/s, Delay: 25 ms, Variance: 10 ms and Distribution: normal

will only be multiplied by a factor of 0.78 [8]. Again, the sender needs the help of the receiver to realized that a packet is lost. Normally the receiver sends an SACK with the bitmask of which packets are lost or sends a packet with a duplicated Acknowledgment (ACK), to notify the sender.

The receiver sorts all packets by its sequence number, to keep the data integrity. However, a malicious peer can save all packets it receives sequentially. This will prevent a gap in the input buffer which is an indicator of a packet loss.

A SACK packet only will be sent, when there is something wrong with the input buffer. Since the input buffer of the modified receiver is always fine, there is no need to send a SACK packet. To inform the sender about the successful acknowledgment of packets, we always send him a SACK message with the information that we received all packets. The sender never decreases the sending rate, since the receiver is misinforming the sender. This significantly increases the sending rate of the sender. Figure 2 shows that the normal file transfer took about 187 seconds. With this attack the file transfer is complete in about 60 seconds. We also increased the average value up to 3414.365 packets/sec. This corresponds to an increase up to a factor of three. The lazy optimistic ack attack provides the foundation for the next attack.

3) Attack 3: Optimistic Acknowledgment: The idea behind the optimistic acknowledgment (opt-ack) attack

is to acknowledge in-flight packets. When the sender receives an ACK which fits into the window, it will decrease the `cur_window` by the size of the payload from the packet which gets acknowledged. The lower the value of `cur_window` is, the more packets the sender is able to send. To acknowledge as much as possible, we initiated the bitmask of a SACK packet with `UINT_MAX` instead of `0`. All bits are set to one, which means we pretend that we received all packets. Even packets that the sender has not sent yet. However, until the acknowledgment arrives at the sender, the sender sends new packets which will automatically acknowledged.

The effect is illustrated in Figure 2. The curve of the opt-ack attack takes an average value of 37.6 s. This is one-fifth compared with a file transfer of a normal receiver. The opt-ack attack produces an average value of 5073.7 packets per second. Consequently this means that the opt-ack attack increased the bandwidth consumption up to five times. The next section will discuss the impact of the presented attacks.

C. Impact of these Attacks

As mentioned in Section III-A, there are two scenarios: Steal Bandwidth or Congestion Collapse. Which of the described attacks can create what damage? To respond to this question we started a file transfer of a 300 MiB file via uTP without any delay in our 100 Mbit/s network. Shortly after that, we started a constant UDP stream of 50 Mbit/s with iPerf [3] for 50 seconds. At first we used the unmodified receiver followed by the modified receiver from the attacks.

Figure 3 shows packets per second depending on the time. Figure 3(a) depicts that, when the UDP stream starts, the uTP stream immediately reduces its sending rate to prevent congestion and to give the foreground traffic a higher priority. Because the delay attack only works when there is a delay and the additional bandwidth consumption is not so high, the Figure would look the same as in Figure 3(a). The Figure 3(b) shows the same experiment with the lazy opt-ack attack. The sender does not recognize the packet loss and this yields to a short congestion between 45–55 s. This corresponds to a packet loss from the constant UDP stream of 7 %. Figure 3(c) shows the results from the experiment with the opt-ack attack. Between the 12th second and 37th second there is no data from both streams. This corresponds to a packet loss from the UDP stream of 42 %. Only when there is a connection timeout from uTP, the constant UDP stream can send data packets to its destination again. The experiment shows that the delay attack can steal additional bandwidth from the sender. With the lazy opt-ack and the opt-ack attack hence it is possible to create congestion. This leads us to the question; what are the limitations of these attacks?

[3] http://sourceforge.net/projects/iperf/

D. Comparison of the proposed Attacks

Figure 4 shows all attacks based on the bandwidth in Mbit/s. All values are average values from ten iterations. The first attack increases the bandwidth up to 1 Mbit/s. This attack can steal additional bandwidth. The lazy opt-ack attack yields a three fold increase in bandwidth. Without a notification of the packet loss, the sender cannot reduce the window size. The opt-ack attack is even more successful and yields a fivefold increase in bandwidth. Both the lazy opt-ack and the opt-ack attack can create congestion.

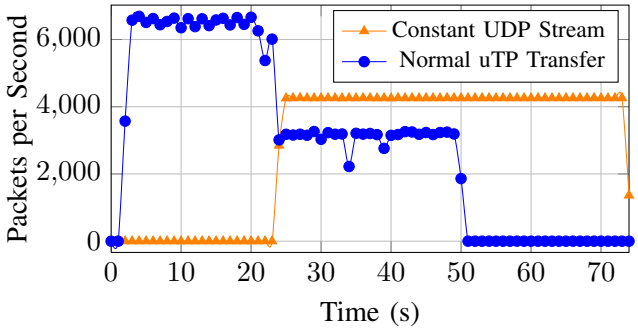
The limitation of these attacks depend on the send buffer of the sender. For the test program `utp_send` this limit is set to 30.000 Bytes. This is the maximum window size of bytes in flight. The average window size of a normal file transfer is up to 17315.7 Bytes.

IV. Countermeasures

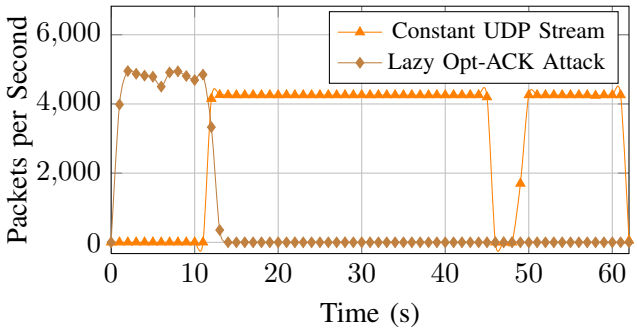
All presented attacks in this paper are hard to detect. This is due to the fact that all attacks differ slightly from a normal behaving receiver. Therefore it is important to have a countermeasure which is efficient and robust to defeat those attacks. In this section we present a countermeasure against the proposed lazy opt-ack and opt-ack attack. We have evaluated the countermeasures and show its severity in terms of performance.

A. Randomly skipped packets

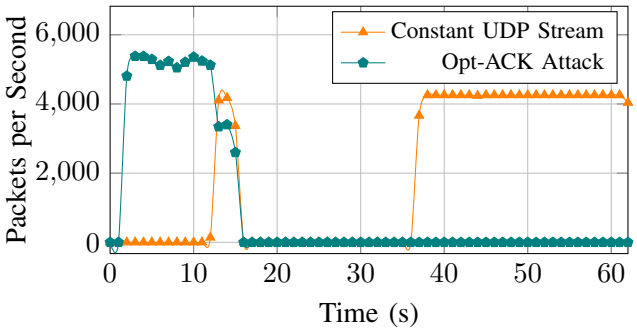
The solution is firstly mentioned by Sherwood et al. [14]. The sender randomly skips packets and remembers which packets he skipped. A normal receiver recognizes a gap in his input buffer and will notify the sender about a missing packet. The receiver does that by not acknowledging that packet or in form of a SACK packet. The sender will then



(a) Without Attack



(b) With the Lazy Opt-ACK Attack



(c) With the Opt-ACK Attack

Figure 3. File Transfer of a 300 MiB File via uTP and parallel to that, a constant UDP Stream of 50 Mbit/s under the following network conditions: Bandwidth: 100 Mbit/s half duplex and Delay: 0 ms

retransmit this packet. An attacker who makes use of the lazy opt-ack or opt-ack attack does not have that gap. This means an attacker will acknowledge the randomly skipped packet nevertheless. An attacker betrays itself through that acknowledgment. The sender check if he receives a ACK which he has not sent out yet. Our implementation of this patch used a random value beginning with 500–700 ms. These random time were experimentally determined. After

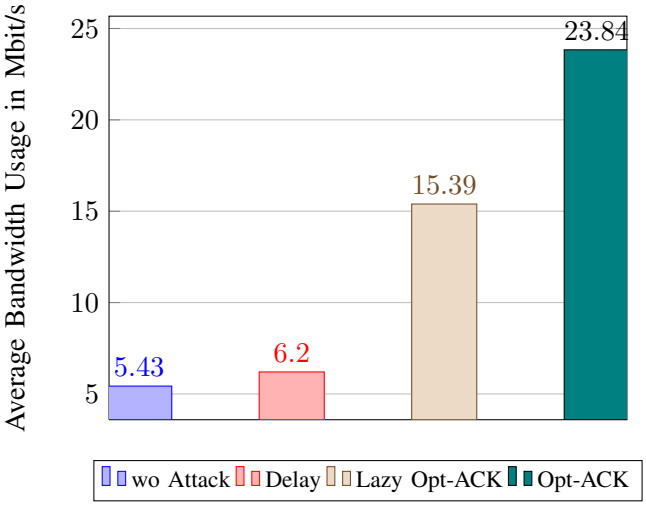


Figure 4. Comparison of all attacks based on the produced bandwidth. All values are average values from 10 iterations.

the sender recognized an attack he will immediately reset the connections. The next section will show the performance of this countermeasure.

1) Performance Evaluation: To evaluate the performance lost of the countermeasure where the sender randomly skips packet, we setup the following experiment: We wrote a Perl script which automatise the file transfer of a 100 MiB file over uTP. The script increases the delay for every iteration by one with a variance of 10 ms and a normal distribution. Every delay value will be tested ten times and the average value will be taken. We repeated the experiment with a normal sender and with a sender which includes the countermeasures.

Figure 5 shows that the difference between a normal sender and patched sender is minimal. So we took the difference of all values and calculated the average value. The average performance lost of the randomly skipped packets countermeasure is 0.448 Mbps. Apparently the longer the delay is, the smaller the difference between the performance of a normal sender and a patched sender. The biggest difference is 2.002 Mbps by 1 ms and the smallest difference is 0.015 Mbps by 24 ms delay.

B. Delay Attack

A countermeasure against the delay attack is more difficult. An attacker can always lie about the one way delay measurement to induce the sender to send

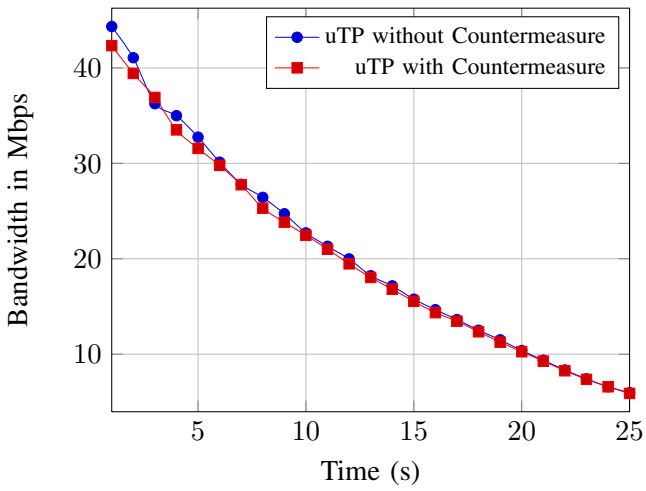


Figure 5. File Transfer of a 100 MiB File via uTP under the following network conditions: Bandwidth: 100 Mbit/s, Delay: 25 ms, Variance: 10 ms and Distribution: normal

more packets into the network. That is because the congestion control with the one way delay only works with the help of the receiver and there is no guarantee that the receiver tells us the truth. One idea to make the window calculation less depending on the one way delay is to include the Round-Trip Time (RTT) into the calculation from Listing 1. This requires that the countermeasure from Section IV-A is included, because the opt-ack attack also reduces the RTT. Therefore a good countermeasure against the delay attack yields to a complete redesign of the LEDBAT congestion control. That is why we don not propose a countermeasure against this attack and point here to further research.

V. Related Work

In this section we discuss the related work which has influenced and inspired this paper. We have divided the related work in a subsection which concerns about the exploitation of congestion avoidance algorithm's and a subsection which concerns about performance evaluations of LEDBAT.

A. Exploit Congestion Avoidance Control

Savage et al. [15] used a misbehaving TCP receiver to get better end-to-end performance. They named one of their techniques *Optimistic ACKing*: The receiver sends ACKs which the sender has not yet been sent. This increases the RTT of the sender and in turn this increases the send rate. Savage et al. showed through their research work that these techniques can increase end-to-end performance. They mentioned that these techniques can also be used to generate a DoS attack.

Sherwood et al. [14] were the first who investigated these techniques from a DoS perspective. They showed that the opt-ack attack can cause widespread damage and destabilize the network. The main difference from Sherwood's opt-ack attack against TCP and the opt-ack attack against uTP is, that uTP does not use the RTT to adjust the send rate. However, every valid ack increases the window size of the sender. As a result of this increase the sender can send new data, which increases the bandwidth. They also engineered a defence strategy which does not need any modification of the TCP/IP standard. Unfortunately this patch has not found its way into the Linux kernel [16]. In contrast, our work provide a performance evaluation of their defence strategy to show the real performance lost.

B. LEDBAT Performance

Rossi et al. [17], [18] were the first who evaluated LEDBAT under scientific aspects. They tested LEDBAT especially in the terms of fairness against competing TCP flows and protocol efficiency. They showed through experimental evidences that LEDBAT reached some of its design goals e.g., protocol efficiency, but they also found some problems regarding the fairness of resources.

Abu and Gordon showed the impact of delay variability on LEDBAT performance caused by router changes [19]. They come to the conclusion that delay variability can cause a negative impact on the throughput.

VI. Conclusion and Future Work

This paper proposed the following three attacks for the uTP: delay attack, lazy opt-ack attack and opt-ack attack. We have shown a detail evaluation and shown the severity of these attacks in terms of bandwidth consumption and number of packets. We have also shown that the delay attack can steal additional bandwidth and the lazy opt-ack and opt-ack attack can cause serious congestion. To better understand bandwidth attacks in general and the proposed uTP in particular, we plan to build a BitTorrent testbed system and repeat our experiments to answer the following questions: How

does the uTP attacks behave in correlation with BitTorrent's choking algorithm? How will the BitTorrent eco-system react on those attacks.

REFERENCES

[1] A. Kuzmanovic and E. Knightly, "TCP-LP: low-priority service via end-point congestion control," *IEEE/ACM Transactions on Networking*, vol. 14, no. 4, pp. 739–752, August 2006.

[2] A. Venkataramani, R. Kokku, and M. Dahlin, "TCP Nice: A mechanism for background transfers," in *5th Symposium on Operating Systems Design and Implementation (USENIX OSDI)*. USENIX, 2002, pp. 329–343.

[3] G. Hazel, "Announcements: μTorrent 1.9 alpha 15380," uTorrent Forum, Nov 2008. [Online]. Available: http://forum.utorrent.com/viewtopic.php?pid=377209

[4] R. Bennett, "Bittorrent declares War on VoIP, Gamers - The Next Internet Meltdown," 2008. [Online]. Available: http://www.theregister.co.uk/2008/12/01/richard_bennett_utorrent_udp/

[5] I. Lamont, "BitTorrent Calls UDP Report "Utter Nonsense"," 2008. [Online]. Available: http://tech.slashdot.org/story/08/12/01/2331257/bittorrent-calls-udp-report-utter-nonsense

[6] I. van Beijnum, "μTorrent's switch to UDP and why the sky isn't falling," 2008. [Online]. Available: http://arstechnica.com/old/content/2008/12/utorrents-switch-to-udp-and-why-the-sky-isnt-falling.ars

[7] S. Shalunov and G. Hazel, "Internet-Draft: Low Extra Delay Background Transport (LEDBAT)," IETF, Tech. Rep., 2011. [Online]. Available: http://tools.ietf.org/html/draft-ietf-ledbat-congestion-09

[8] A. Norberg, "BEP 0029: uTorrent transport protocol," BitTorrent, Inc., Tech. Rep., 2010. [Online]. Available: http://bittorrent.org/beps/bep_0029.html

[9] V. Jacobson, "Congestion Avoidance and Control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314–329, August 1988.

[10] F. Adamsky, H. Khan, M. Rajarajan, and S. A. Khayam, "POSTER: Destabilizing BitTorrent's Clusters to Attack High Bandwidth Leechers," in *ACM Computer and Communications Security 2011*, 2011. [Online]. Available: http://florian.adamsky.it/research/publications/2011/ccsp152a-adamsky.pdf

[11] A. Legout, N. Liogkas, E. Kohler, and L. Zhang, "Clustering and sharing incentives in BitTorrent systems," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 2007, pp. 301–312.

[12] E. Van Der Sar, "uTorrent Keeps BitTorrent Lead, BitComet Fades Away," TorrentFreak, September 2011. [Online]. Available: http://torrentfreak.com/utorrent-keeps-bittorrent-lead-bitcomet\-fades-away\-110916/

[13] S. Hemminger, "Network emulation with NetEm," in *Linux Conf Au*. Citeseer, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf

[14] R. Sherwood, B. Bhattacharjee, and R. Braud, "Misbehaving TCP receivers can cause Internet-wide congestion collapse," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 383–392. [Online]. Available: http://portal.acm.org/citation.cfm?id=1102120.1102170

[15] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 71–78, October 1999.

[16] S. Hemminger, "Re: fixing opt-ack DoS against TCP-Stack," Linux-Net Archive, Jan 2007. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/net/0701.1/0003.html/

[17] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: the new BitTorrent congestion control protocol," in *Computer Communications and Networks (ICCCN)*. IEEE, 2010, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5560080

[18] D. Rossi, C. Testa, and S. Valenti, "Yes, we LEDBAT: Playing with the new BitTorrent congestion control algorithm," in *Passive and Active Measurement*. Springer, 2010, pp. 31–40. [Online]. Available: http://www.springerlink.com/index/M4V0200776L3P52H.pdf

[19] A. J. Abu and S. Gordon, "Impact of delay variability on LEDBAT performance," in *Advanced Information Networking and Applications*. IEEE, 2011, pp. 708–715. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5763498