Estimating Congestion Window with Misbehaving Client using Proposed FACK

Ramesh K
P.G. Department of Computer Science,
Karnataka University, Dharwad, Karnataka

Abstract: Detect and fix a misbehaving TCP implementation/client that is trying to get more than its fair share of bandwidth by artificially acknowledging not received packets in order to advance the senders window and thus obtain better bandwidth. This papers talks on the method which relays on the RTT estimate to detect if the receiver is misbehaving and if so proposes a solution to punish the misbehaving client.

SACK provides the mechanism to recover multiple packet loss with cumulative acknowledgement. Though the SACK helps in recovery of multiple packet loss, but it's not designed to address congestion issues. FACK uses the information from SACK to inject packet into network more precisely during congestion recovery. The core of FACK congestion algorithm is the information on the forward-most data held by the receiver. This information is gathered from the receiver produced ACK. So in case of DOS (misbehaving client) where the client intentionally sends the FACK for the packet it is not received. Which increase the FACK window and number of packet injection into the network by the sender which intern result in the congestion and client will be getting more than fair share of bandwidth We introduce some checks at the sender based on RTT to control this kind of behavior and punish the clients.

1. Introduction

TCP/IP is the ubiquitous Internet protocol that forms an integral part of the communications infrastructure. It has many parameters that help provide guaranteed reliability, congestion control, rapid data flow increase to utilize available bandwidth among other features. However, some of these features can be abused by unscrupulous clients to usurp more than their fair share of bandwidth to the detriment of other clients. Described is one of these scenarios and propose a solution to prevent abusers from exploiting TCP features to get more than their fair share of bandwidth.

TCP operates using a sliding window protocol where data to the left of the window has been acknowledged and data to the right of the window has not been sent yet. The window is the region of interest for the sender and as the receiver ACKs the data in that region, the window moves to the right and more data is sent. Otherwise, the sender may have to retransmit the data if the ACK has not been received within a certain time limit (a function of the estimated round trip time from the sender to receiver).

2. SACK and FACK

With the cumulative acknowledgment scheme used by TCP, the sender can only learn about a single lost packet per round trip time. This is a great limitation in the case of multiple losses in a window of data, its consequence being poor performance. Through the Selective Acknowledgment (SACK) mechanism the data receiver informs the sender about all the segments that have arrived successfully, so the sender only needs to retransmit only segments that have actually been lost. The Forward Acknowledgment (FACK) algorithm is designed to be used with the SACK option. FACK uses the additional information provided by SACK to keep an explicit count of the total number of bytes of data outstanding in the network. The FACK option provides an improvement in performance in the case of multiple losses in a single window of data and reduces the overall burstiness of TCP.

3.FACK Design Goals

Under single segment losses, Reno implements the ideal congestion control principles set forth above. However in the case of multiple losses, Reno fails to meet the ideal principles because it lacks a sufficiently accurate estimate of the data outstanding in the network, at precisely the time when it is needed most.

The requisite network state information can be obtained with accurate knowledge about the forward-most data held by the receiver. By forward-most, we mean the correctly received data with the highest sequence number. This is the origin of the name \forward acknowledgment." The goal of the FACK algorithm is to perform precise congestion control during recovery by keeping an accurate estimate of the amount of data outstanding in the network. In doing so, FACK attempts to preserve TCP's Self-clock and reduce the overall burstiness of TCP.

4.The FACK Algorithm

The FACK algorithm uses the additional information provided by the SACK option to keep an explicit measure of the total number of bytes of data outstanding in the network. In contrast, Reno and

Reno+SACK both attempt to estimate this by assuming that each duplicate ACK received represents one segment which has left the network. The FACK algorithm is able to do this in a straightforward way by introducing two new state variables, snd:fack and retran_data. Also, the sender must retain information on data blocks held by the receiver, which is required in order to use SACK information to correctly retransmit data. In addition to what is needed to control data retransmission, information on retransmitted segments must be kept in order to accurately determine when they have left the network.

At the core of the FACK congestion control algorithm is a new TCP state variable in the data sender. This new variable, snd:fack, is updated to correct the forward-most data held by the receiver. In non-recovery states, the snd:fack variable is updated from the acknowledgment number in the TCP header and is the same as snd:una. During recovery (while the receiver holds non-contiguous data) the sender continues to update snd:una from the acknowledgment number in the TCP header, but utilizes information contained in TCP SACK options 6 to update snd:fack. When a SACK block is received which acknowledges data with a higher sequence number than the current value of snd:fack, snd:fack is updated to re ect the highest sequence number known to have been received plus one.

Sender algorithms that address reliable transport continue to use the existing state variable snd:una. Sender algorithms that address congestion management are altered to use snd:fack, which provides a more accurate view for the state of the network.

We do need awnd to be the data sender's estimate of the actual quantity of data outstanding in the network. Assuming that all unacknowledged segments have left the network:

awnd = snd:nxt -- snd:fack **(1)**

During recovery, data which is retransmitted must also be included in the computation of awnd. The sender computes a new variable, retran_data, correcting the quantity of outstanding retransmitted data in the network. Each time a segment is retransmitted; retran_data is increased by the segment's size; when a retransmitted segment is determined to have left the network, retran_data is decreased by the segment's size. Therefore TCP's estimate of the amount of data outstanding in the network during recovery is given by:

awnd = snd:nxt-- snd:fack + retran data **(2)**

Using this measure of outstanding data, the FACK congestion control algorithm can regulate the amount of data outstanding in the network to be within one MSS of the current value of cwnd:

```
While (awnd < cwnd)
                    sendsomething();
```

The FACK congestion control algorithm does not place special requirements on sendsomething(); the algorithm implied by the SACK Internet-Draft is su cient. Generally sendsomething() should choose to send the oldest data first.

FACK derives its robustness from the simplicity of updating its state variables: if sendsomething() retransmits old data, it will increase retran data; if it sends new data, it advances snd:nxt. Correspondingly, ACKs which report new data at the receiver either decrease retran data or advance snd:fack. Furthermore, if the sender receives an ACK which advances snd:fack beyond the value of snd:nxt at the time a segment was retransmitted (and that retransmitted segment is otherwise unaccounted for), the sender knows that the segment which was retransmitted has been lost.

5.The Problem

Consider the following scenario where a misbehaving receiver (client) sends some forward ACK for some data it has not received. At that point in time it may be data that the sender may not have sent or the sender may have just sent. Also, the client probably does not care about the data (or is willing to take the chance of losing it just so that it can increase its bandwidth more rapidly than others) since it is willing to falsely acknowledge it. However, the sender on receiving the ACK will increase the window size (thus the bandwidth) and may even get a skewed (reduced) estimate of the RTT(round trip time) which will force it to transmit faster for this client and will unfairly impact others (produce more congestion while unfairly giving an advantage to this client).

We propose a solution to detect and fix a misbehaving TCP implementation/client that is trying to get more than its fair share of bandwidth by artificially acknowledging not received packets in order to advance the sender's window and thus obtain better bandwidth. This invention relies on the RTT estimate to detect if the receiver is misbehaving and if so proposes a solution to punish the misbehaving client.

6.Our Solution

Proposed is the following solution to solve this problem. After receiving an ACK for a packet, the sender measures the RTT (round-trip-time) for it. If the RTT is below a certain threshold (abnormally low RTT- this can be based on the average RTT for example) the following steps are taken:

1) The first time this occurs, it may be an anomaly to ignore but mark that this happened so that if it happens again, it is a sign that some client may be abusing and the server can then take remedying action. Also this value will not be used in computing the new estimated RTT so there isn't a skewed estimate.

2) Set a user-tunable threshold for the number of times this will be allowed to happen and if this threshold is crossed remedying action is taken.

3) If threshold is exceeded, punish the client by reducing its bandwidth. This can be done by a) not increasing the window size OR b) by increasing the window size only after the sender's RTT estimate has expired.

4) If client continues to abuse repeatedly, the server administrator may wish to block access to the abusive client. If so, capability to blacklist this client is provided and discontinues service to this client if so desired.

By taking these (and other possible) measures the TCP sender can prevent unfair clients from prematurely ramping up their window sizes and hence unfairly deriving more bandwidth through false ACKing.

7.Proposed FACK Algorithm

FACK is entirely new algorithm which uses additional information provided by the SACK to calculate total number of outstanding data on the network. It introduces two new variable 'snd.fack' and 'retran_data' to help in estimating number of packets to be injected in to network.

In TCP implementation 'snd.una' holds the sequence number of first byte of unacknowledged data and 'snd.nxt holds the sequence number of first byte of unsent data. In non-recovery states, the snd.fack is updated from the acknowledgement number in the tcp header and is same as snd.una. During recovery when receiver holds non-contiguous data sender continue to update snd.una from the acknowledgement number but utilizes information contained in TCP SACK to update the snd.fack.

During recovery snd.fack reflect the highest sequence number updated from the SACK plus one. Now to calculate actual data outstanding in the network is
Outstanding data = last acknowledge data sequence – highest sequence number at SACK+ 1

i,e ownd = snd.nxt – snd.fack ---------- **(1) .**

During recovery retransmitted data also should be considered as they are already injected into network

ownd = snd.nxt – snd.fack + retran_data --------- **(2).**

Using these measurements FACK can regulate the amount of data to be injected in to the network nearly as below.

```
While (ownd < cwnd)
 {
        if ( retransmit time out )
                retransmit data
        retran_data  =+ retransmit data
        else if ( send  new data )
                snd.nxt =+ newdata size
 }
```

In the similar way, if the new SACK received which either decrease the retran_data or advance the snd.fack.

So snd.fack and retran_data variable are controlled by the SACK data provided by the receiver. If a receiver is designed for DOS and not care about the packet it receives. It can falsely advance snd.fack by acknowledging the packet it have not received yet or the packet which is not sent yet, which results in false calculation of outstanding data in the network(lesser than the actual). Below we discuss the different cases and propose the solution for the same.

8.Test Cases

Case 1: If the acknowledgement received for the packet which is less than snd.nxt

```
If (RTT < = avgRTT/2)          *1
 {
```

- Mark this event
- Do not use this RTT value for avgRTT calculation

```
Counter=+1
If (counter > usr specified limit) *2
    {
        wait till RTT => avgRTT then calculate ownd      *3
    }
```

Case 2: If the acknowledgment is received for the packet which is not yet sent.

```
If  ( snd.fack  >  snd.nxt )

   {
                - Mark this event
                - Do not use this RTT value for
                  avgRTT calculation
        `          Counter=+1
        If ( counter > usr specified limit ) *2
        {
wait till RTT => avgRTT then calculate ownd *3
        }
   }
```

*1 Though the RTT can be attacked but one cannot make it half of its real value as in real at least one way communication time cannot be intruded.

*2 User can set the threshold for number of time this type of even can be allowed.

*3 wait until avg RTT for this action then calculate the ownd as specified at formula 2.

9. Conclusion

The proposed FACK algorithm (test cases) detect and fix a misbehaving TCP implementation/client that is trying to get more than its fair share of bandwidth by artificially acknowledging not received packets in order to advance the sender's window and thus obtain better bandwidth. This invention relies on the RTT estimate to detect if the receiver is misbehaving and if so proposes a solution to punish the misbehaving client. This paper has some of the advantages compared to that already existed FACK algorithm.

It helps in recognizing the misbehaving clients and punish them with user specified way and also helps in calculating exact number if outstanding data in the network there by not letting to increase congestion during congestion time.

References

1. Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, August 1999.
2. Sally Floyd. TCP and successive fast retransmits. http://www.aciri.org/floyd/papers/fastretrans.ps,May 1995.
3. Reza Rejaie, Mark Handley, and Deborah Estrin.R
4. AP: An end-to-end rate-based congestion control mechanism for real-time streams in the Internet. In *INFOCOM '99*.
5. Scott Shenker. Making greed work in networks: A game-theoretic analysis of switch service disciplines. In *SIGCOMM '94*, pages 47–57, August 1994.
6. Je C. Mogul. Observing TCP Dynamics in Real Networks. Proceedings of ACM SIGCOMM 92, pages 305{317, October 1992.
7. J. Postel. Transmission Control Protocol, September 1981. Request for Comments 793.[Ste94]
8. W. Stevens. TCP/IP Illustrated, volume 1. Addison-Wesley, Reading MA, 1994.
9. W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, March 1996. Currently an Internet Draft: draft-stevens-tcpca-spec-01.txt.[tcp95]
10. Minutes of the tcp x meeting at the 34th IETF, in Dallas TX, December 1995. Obtain via: http://www.ietf.cnri.reston.va.us/proceedings/95dec/tsv/tcplw.html. [ZSC91]
11. Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Transmission. Proceedings of ACM SIGCOMM91, pages 133{148, 1991
12. Van Jacobson. Congestion Avoidance and Control. Proceedings of ACM SIGCOMM '88, August 1988.[Jac90]
13. Van L. Jacobson. Fast Retransmit. Message to the end2end-interest mailing list, April 1990.[Jac95]
14. Van Jacobson, July 1995. Private Communication.[JB88]
15. V. Jacobson and R. Braden. TCP extensions for long-delay paths, October 1988. Request for Comments 1072