Application Centric Autonomic BW Control in Utility Computing

Krishna Kant
Intel Corporation

***Abstract*— QoS and congestion performance are crucial to good application performance in a utility computing environment. Unfortunately, proper IP QoS setup is very complex and is either ignored completely or set rather simplistically. It is well known that without an elaborate end to end QoS setup, TCP connections simply divide up the available excess bandwidth equally among themselves under congestion. In this paper we propose autonomic mechanisms that determine BW requirements of various flows of an application and maintain them in appropriate proportion even during congestion. The estimations are done dynamically and thus can easily track changing application requirements. The paper shows that the scheme not only yields close to desired bandwidth allocation, but also significantly reduces packet losses.**

Keywords: Utility computing, TCP, QoS, congestion control, collective control, diffserv

I. Introduction

Utility computing involves running large applications consisting of many processes spread across a number of nodes. These nodes could be confined to a local environment (e.g., cluster computing), geographically distributed (e.g., grid computing), or a combination of the two (a grid of clusters). Utility computing also encompasses consolidation and dynamic resource management in form of virtual clusters. In this paper, we consider the issue of autonomic BW control for applications in such an environment under congestion conditions. An application (or a related group of applications) consists of multiple *inter-related* "flows". Under congestion conditions, good application progress often demands a coordinated control of the flows (e.g., a proportional allocation of BW to various flows) rather than an independent one (which is the default behavior of reliable transport layer). The traditional IP QoS does provide a rich arsenal of techniques for BW, latency and jitter control, and can, in theory, be harnessed to achieve coordination. In particular, the diffserv setup at the routers can be used to limit flow bandwidths and do weighted fair queuing to achieve proportional flows. Unfortunately, such a control is impractical, tricky to administer [4] and requires detailed application knowledge which is rarely available. Because of this, IP QoS parameters in reality are either left at their default values, or set up for a coarse grained differentiation (e.g., giving higher priority to control or streaming traffic).

We believe that to be successful, any manual QoS setup must remain "common-sense", and more sophisticated treatment should be autonomic in nature. In particular, for SLA purposes, there should be no need to specify any more about the application flows than their general type (e.g., real-time vs. non real-time, normal vs. premium, etc.). The autonomic mechanism should automatically learn the application characteristics, bandwidth requirements, etc. and then use some sort of a signaling mechanism to setup and evolve the QoS settings. Undoubtedly, this is a very tall order and we do not pretend to solve the entire problem here. We do, however, propose some simple schemes in this paper towards this goal and exhibit their performance.

The outline of the paper is as follows. In section II we discuss how the problem and work in this paper differs from previous work on the subject. In section III we describe essential elements of collective bandwidth control mechanism and provide some insight into various decisions made in choosing the algorithms. Section IV then discusses the experimental setup for evaluating the scheme and the results. Finally, section V concludes the discussion.

II. Utility Computing Environment

We start with a brief characterization of utility computing environments in terms of QoS needs. As stated above, utility computing subsumes both cluster and grid computing models. In the cluster computing context, there is an emerging trend for Ethernet as a single *unified fabric* that carries all traffic types, each with its own QoS needs. In a grid computing environment, by necessity, the communication bandwidth requirements are typically lower and latency tolerance much greater; however, the QoS issues become even more important because of a large number of intervening nodes and unknown cross traffic.

Excluding streaming media, the application "flows" typically correspond to TCP connections. In case of congestion, the appropriate response is typically to squeeze all flows proportionately. For example, suppose that a distributed DBMS generates 50 Mb/sec storage traffic and 20 Mb/sec IPC traffic. Now, under congestion if IPC traffic can only drive 10 Mb/sec, there is little point in providing more than 25 Mb/sec to the storage traffic. Similar arguments can be made regarding set of flows of different applications or even groups of applications. As an example, a data mining and a data rendering application may be working in tandem to perform the desired higher level function and thus will work best with proportional degradation.

Unfortunately, TCP doesn't behave this way and will try to equalize bandwidth allocations. Thus, the min-max fairness property of TCP is not desirable from a distributed application perspective. In view of difficulties in harnessing IP QoS to force "proportional fairness", the problem must be solved at the transport layer itself.

Accordingly we introduce the notion of *collective bandwidth control* or more generally *collective resource control* so that all flows passing through a congestion point are controlled as a cooperating set, rather than as unrelated competing set. This collective control could be operating at any of the 3 levels indicated above (inter-VC, inter-application and intra-application). For the purposes of the paper, we assume that relevant flows use a reliable connection protocol such as TCP, although similar techniques can be used above the transport layer for UDP flows as well. We also assume that all flows are reasonably long lived, so as to make their autonomic control meaningful. This is a reasonable assumption for many utility computing applications.

Before continuing, we briefly review the related work. TCP and TCP-like congestion control is an extremely well researched area and we will not try to survey it. In particular, numerous loss and delay based schemes have been investigated of which the latter are particularly relevant here [1], [3]. Mathematical modeling of TCP behavior including fairness, stability, Nash equilibria, is another heavily studied area [5], [2], [4]. Enforcing weighted proportional fairness [6] is close to the ideas explored in this paper, except that our focus is on applications rather than individual flows with statically specified requirements. Aggregated control of BW is explored in the context of parallel TCP in [9], [8] where the windows of all parallel flows are controlled similarly in response to loss/congestion experienced by one connection. The purpose of this work isn't proportional fairness, but rather to make use of scarce loss information for the entire group of parallel connections. Finally, although this paper refers a lot to bandwidth allocation, it has little relationship to the literature on equivalent bandwidth, VBR BW estimation and the like. Instead, we track BW requirements dynamically as the application evolves.

III. Collective Bandwidth Control

There are 3 essential aspects of collective bandwidth control: (a) determining all relevant flows (or connections in this paper) that need to be controlled together, (b) determining bandwidth requirements of such connections, and (c) starting and ending control. For simplicity, we henceforth assume that all flows are permanent. The schemes can be easily extended to non-permanent flows provided that the flows are sufficiently long to make per flow monitoring sensible.

A. Determining Collective Control Set

The control set determination problem must be addressed independently by each *physical* server since a coordinated determination, while theoretically feasible, would usually be too complex. The determination really involves estimating which connections from this server are affected by the existing congestion so that they could be controlled cooperatively. This is difficult to do purely from the default TCP behavior since there is no guarantee that all affected connections will receive ECN (explicit congestion notification) or experience packet loss. In fact, with a RED-like marking/dropping scheme, fatter connections are more likely to experience congestion indication.

The collective control set (CCS) can be identified in many ways, each with its pros and cons. Ideally, the CCS consists of all connections that pass through the congested router(s). Unfortunately, existing protocols don't provide any means to identify the congested device or the connections passing through them. Therefore, this needs to be "approximated". One simple, but not very accurate, approximation is to group together all connections to the same destination. A more sophisticated (but somewhat impractical) method is to set up routers for 100% ECN marking so that all affected connections are revealed quickly. A yet another method is to observe the performance of connections (e.g., RTT and window contraction) and from there try to "guess" which connections are experiencing congestion. A detailed comparison of various schemes for determining CCS is beyond the scope of this paper. Here we show the effectiveness of collective control by assuming that we have somehow determined the CCS already.

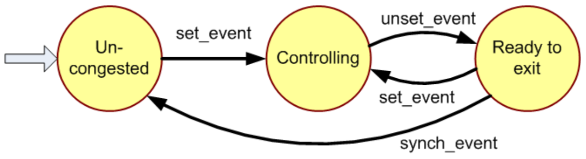

Fig. 1. An illustration of bandwidth estimation process

B. Operation of Control Cycle

As stated earlier, our scheme operates by autonomically estimating bandwidth requirements and other parameters of the connections and using them for collective resource control. At a high level, the control cycle involves the following three components:

(a) Detection of when the first member of a CCS starts encountering congestion so that the scheme can move from per connection control phase to collective control phase.
(b) Reliable estimation of the bandwidth and other relevant parameters (e.g., delay).
(c) Reliable estimation of when the CCS is free of congestion, so that a switchover back to individual control can be effected.

Fig. 1 shows these components in form of a generic state diagram. Initially, a connection is in the uncongested state and is running the estimation algorithms. The "set_event" refers to situations that result in the connection moving to a state where collective bandwidth control (CBC) goes into effect. The "unset_event" refers to the situation where this connection does not see any congestion effects and thus declares itself ready to exit from CBC regime. However, it will still be subjected to CBC until all connections in CCS are ready to exit. The "synch_event" indicates this condition and makes the process start over.

The development of control algorithm in the following introduces several parameters, however, none of these is intended to be a "tunable" parameter since that would defeat the purpose of an autonomic scheme. Instead, the parameters are estimated from detailed simulations and expected to stay constant in actual implementations.

1) Setting Controls: It is well known that TCP regulates flow of packets in response to two conditions: (a) explicit congestion indication via ECN or packet loss, and (b) increase in round-trip delay. Different versions of TCP may weight these events differently; for example, TCP vegas and the newer fast TCP [1], [3] base their congestion control algorithm directly on the observed delays. Thus, "set_event" should also comprehend delay, ECN and loss. The last two are easy to handle: the CBC takes effect whenever TCP slashes window as a result of ECN processing, retransmission timer expiring, and fast retransmit. For the delay, we keep track of two quantities over the last N_{rtt} round-trip periods (where N_{rtt} is a parameter).

1) current_rtt. Average RTT over the last N_{rtt} measurements done by TCP. The averaging over multiple measurements is done to smooth out short-term variations.
2) base_rtt. This is an exponentially smoothed version of current_rtt with smoothing factor of α_r. This estimate is required to detect congestion onset and is frozen as soon as the connection enters the "Controlling" state.

Given these estimates, the increase in delay is detected when ratio of current_rtt and base_rtt exceeds some threshold β. The threshold β must be chosen large enough so that CBC goes into effect only for real congestion situations rather than due to variations in queuing and scheduling delays. For the results shown here, we chose $\beta = 1.4$. We studied the performance with a range of values for β and found that with $\beta < 1.25$, the control may take effect spuriously. A larger value is safer, but delays the control in case of real congestion.

The choice of the parameter N_{rtt} needs to balance two conflicting needs: (a) N_{rtt} should be rather small in order to detect RTT increase before the losses take place, and (b) N_{rtt} should be large enough to smooth out bumps due to packet losses. After extensive experimentation, we found that a rather small value of $N_{rtt} = 4$ is adequate in all cases, and is used in the results. The parameter α_r needs to be rather small to allow a substantial difference between current_rtt and base_rtt as delays start to mount. At the same time, α_r should be large enough to produce a good estimate of RTT and bandwidth (see below) in a reasonable amount of time. Based on a number of experiments, we chose $\alpha_r = 0.05$. The results aren't terribly sensitive to the choice though as one might expect.

2) Bandwidth Estimation: In many TCP implementations, the window size "cwnd" is often not a good indicator of used bandwidth; therefore, the bandwidth estimation was done directly. In particular, the following three quantities were estimated (this is similar in spirit to delay estimation discussed above):

1) current_bw. Estimated as the byte acknowledgement rate over the last N_{rtt} RTTs (using TCP's highest unacked sequence number).
2) reqd_bw. This is an exponentially smoothed version of current_bw with smoothing factor of α_r and indicates the required bandwidth. This estimate is frozen as soon as the connection enters the "Controlling" state.
3) avail_bw. This is also an exponentially smoothed version of current_bw but with smoothing factor of α_a. This estimate is updated continuously.

The parameter α_a must necessarily be significantly larger than α_r since the purpose of avail_bw is to faithfully track the impact of ongoing congestion episode. The smoothing is still required, however, because of drastic window cuts inherent in TCP. After some experimentation, we settled on $\alpha_a = 0.2$, which should work reasonably well across a range of circumstances.

3) Removing Controls: As we developed the scheme, we found that control removal is much more challenging than setting it. The first problem is that the effects of the congestion may linger on long after the congestion source has quieted. Thus a premature return to "uncongested" state could quickly go back to congested state with incorrect parameters latched in! A more difficult problem is that shortly after the congestion source goes off, things begin to look normal until the squeezed flows are able to increase their windows to take up the slack. This brief period forms a trap for removing controls that must be avoided. Fortunately, keeping the controls on somewhat longer than necessary has no serious consequences.

The basic triggers for control removal are: (a) drop in current_rtt to the level of base_rtt, and (b) rise in avail_bw to beyond reqd_bw. The accumulated backlog during the congestion period will mean that the application will drive more than reqd_bw for some time – this creates a "hump" in the BW usage. However, this is very much a function of the application behavior. For example, if the application cannot get its IPC traffic through, it will simply slow down without any "backlog" per se. These considerations make control removal somewhat complex. Basically, the scheme includes a provision of "going over the hump" (in cases of backlog), ensuring that control removal is not done for some minimum period following normal behavior (to avoid the trap), and releasing control if no hump is observed. Also, the removal for any given connection simply puts it in "ready-to-exit" state. The actual removal of controls is done only when all connections in the group have migrated to "ready-to-exit" state.

C. Collective Bandwidth Control

Given the required bandwidths of all connections within a CCS (collective control set), the final question is how to make TCP connections divvy up the bandwidth properly. Our goal here was to keep the scheme simple so that we don't break other features of TCP such as fast retransmit, fast recovery, SACK, RTT estimation, etc.

Consider the connection $i \in 1..N$ in the CCS with N connections. A simple way to enforce bandwidth division is by capping the transmit window size. The cap, denoted W_i^{max}, is given by:

$$W_i^{max} = \min(\eta\lambda_a, \overline{\lambda_r}) RTT_{avg} \lambda_{ri} / \overline{\lambda_r} \qquad (1)$$

where η is a parameter (to be discussed shortly), RTT_{avg} and $\overline{\lambda_r}$ are average RTT and required bandwidth over the CCS (e.g., $\overline{\lambda_r} = \sum_{j=1}^{N} \lambda_{rj}/N$). The second term in the equation is basically the average BW-delay product apportioned between various connections in proportion of their requirements. The first term ensures that we don't go below the available bandwidth; in fact, $\eta > 1$ ensures a certain degree of aggressiveness, discussed next.

The parameter η plays crucial role in the control. If $\eta \leq 1$, the controller will be unable to make use of any additional bandwidth that becomes available and thus cannot recover quickly as the interfering traffic goes down or disappears

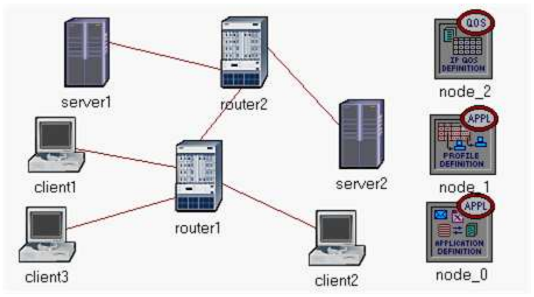

Fig. 2. A network with two database and one FTP applications

completely. In fact, even η slightly larger than 1.0 may make the connection too sluggish to survive under competition from higher priority interfering traffic. On the other hand, a large η implies that the connection will be very aggressive in grabbing additional available bandwidth and thus lead to more severe oscillations in the thorughput. A value of η in the range of 1.15 and 1.25 works reasonably well. The main problem we faced in our experiments was the resetting of database connection when it suffered too many retransmissions because of backlogged traffic. This problem arises because Opnet applications do not enforce any socket level tranmission limits.

IV. Experimental Results

In order to study the behavior of proposed scheme, we consider a simulation setup using OPNET (www.opnet.com). The choice of OPNET makes the simulation quite realistic because OPNET provides implementation of all network layers including Ethernet MAC, IP, TCP, etc. in great detail. Opnet also provides several built-in applications and detailed switch/router models. The next few sections describe the model and its performance under a variety of scenarios. Unfortunately, given the sheer number of parameters at IP and TCP level that one needs to set up and their complex interactions makes it difficult to cover the entire space.

A. Model Architecture

Fig. 2 shows the network model used in the experiments reported here. Server1 hosts two database applications and Server2 hosts an FTP application. Clients 1 and 2 generate queries for DB applications 1 and 2 respectively. Client 3 generates file transfer requests for Server2. All traffic goes via identically configured routers 1 and 2. All links shown are standard 10 Mb/sec FDX (full duplex) with default parameters.[1] The router models try to mimic 3Com CB6000 router features. The 3 nodes on RHS in Fig. 2 are for setting QoS, application profiles, and application definitions respectively. Each client and server nodes is represented by a *node model* (not shown) and includes explicit representation of application, TCP, IP, MAC layers.

Each database application is designed to run over a single permanent TCP connection. (Connection resets under overload were avoided by making TCP retransmission limit very high.) The FTP application, on the other hand, establishes a new TCP connection for each file transfer. The database traffic is 100% query type (i.e., no updates) and thus most of the traffic is from server to client direction. On the other hand, the FTP traffic is split 50-50 between gets and puts. DB traffic from both clients has exponential inter-arrival times and with response sizes uniformly distributed in the range of 8KB and 16KB. The main distinction between clients 1 and 2 is that client 1 drives twice as much response BW as client 2 (4.8 Mb/sec vs. 2.4 Mb/sec excluding about 10% application and lower layer protocol overhead). The FTP requests also have exponential inter-arrival times but involve 25 parallel streams, each requesting files uniformly distributed in the range of 5KB and 35KB. The total *offered* FTP traffic is set at 2 Mb/s, 4 Mb/s and 8 Mb/s (again, excluding about 10% protocol overhead).

[1]10 Mb/s speed was chosen for quick simulations; similar results are obtained w/ higher speed links as well.

It is clear that the link between the two routers is most loaded and we want to examine consequences of this. Also, the intent of the setup is to treat FTP traffic as the “interfering traffic” and examine the performance impact on the database traffic. This also means that *the autonomic monitoring and control discussed here are applied only to the DB traffic*. The FTP traffic comes on at time 30 secs, stays on for 60 secs, and then goes away. The database traffic is always on (starting at time 5 secs to allow for routing table updates to settle down and TCP connections to be established).

In terms of relative priorities, we considered two cases: (a) FTP with higher priority (via DSCP value of AF31), and (b) all traffics at the same priority (via common DSCP value of AF21). The used router model (CB600) did not give give AF31 a strict priority over AF21; instead, it used a queue double in size that for AF21. We maintained this implementation feature in our experiments. In particular, we used 2 different queue sizes: (a) *full queues*, sized at 120 & 60 packets respectively for AF31/21, and (b) *half queues*, sized at 60 and 30 packets for AF31/21. The routers were configured with centralized buffering scheme (shared by all ports) instead of per port buffering. Centralized buffering uses the router memory much more efficiently and is often the default operation mode.

The routers were configured for RED and involve 3 parameters: (a) min threshold $L_{\min}$, (b) max threshold $L_{\max}$, and (c) drop percentage P_D. The basic idea is to start dropping packets when $L_{\min}$ threshold is exceeded, and reach P_D drop percentage when $L_{\max}$ threshold is reached. The queue sizes indicated above are really $L_{\max}$ values indicated above. We also chose $P_D = 10\%$. It is well known that proper setting of RED parameters is extremely difficult [4]; therefore, our settings are rather arbitrary and do not change (except for the 2 queue sizes studied above).

Opnet’s original TCP implementation supports most of the popular variants (e.g., Reno, NewReno, Tahoe), SACK, ECN, etc. We modified it with CBC so that all these features will continue to work. However, this is not to say that the collective control will work the same way under various options. For the purposes of this paper, we kept the congestion control rather simple: we disabled SACK and used Reno version of TCP. As stated earlier, ECN was disabled since it is still not supported by most real routers. Examining performance of collective control under other situations is beyond the scope of this paper.

Opnet’s original TCP version is designated as v3; so, we called ours as TCPv4. The results below compare v3 and v4 under a variety of circumstances. It is important to note that TCPv4 comes into play only when there are competing connections at an endpoint. The receiver buffer was set to 64KB, and TCP timers were adjusted downwards in order to allow high throughput per connection without connection resets. To exhibit TCPv3 vs. v4 performance, we will show the following key parameters in the results that follow:

1) Carried interfering FTP traffic (server to client direction).
2) Carried DB1 and DB2 traffic (server to client direction).
3) Overall dropped traffic at router2. (Router1 traffic is small and does not experience drops).
4) RTT and retransmissions for the TCP connection carrying DB1 traffic (server to client direction). DB2 traffic parameters are generally similar and not shown.

The simulation collects a huge number of other parameters as well (e.g., window size, in-flight size, queuing delays, etc.) but these are not shown to avoid clutter.

B. *Full Queue Results*

Figs 3-8 show comparative performance under offered FTP traffic of 8 Mb/sec (not counting about 10% overhead) and full router queues. Note that this situation corresponds to a severe congestion at inter-router link since we are trying to drive 16.7 Mb/sec traffic through a 10 Mb/sec link.

TCPv3 behavior is shown in red lines (lighter shade in printouts) in all graphs, and is discussed in this paragraph. The severe congestion results in a significant amount of packet losses and retransmissions while TCPv3 attempts to cut down the traffic. The RTT also increases due to congestion. The most interesting behavior is that of DB1 vs. DB2 throughput. It is seen that DB1 traffic is affected severely but DB2 traffic almost stays at the same level! In other words, TCP pretty much equalizes the bandwidth of the two flows even though we started with 2:1 ratio. This is an expected TCP behavior.

Let us now examine the TCPv4 behavior (shown in dotted lines). It is seen that both IP packet drops and retransmissions go down drastically, and the RTT shows much lower fluctuations. All of these are highly desirable from an application perspective. In fact, packet drops happen only initially which points to a more effective congestion control. Also, unlike TCPv3, the DB2 flow also suffers, in a proportion closer to 2:1. As a bonus, the throughput is far less variable than for TCPv3. The overall router link utilization (not shown for brevity) also shows much fewer and shorter excursions below the nominal 100% level. Even the interfering FTP traffic experiences a much smoother throughput for v4. It follows that CBC is working extremely well, although the 2:1 throughput ratio isn't quite achieved (more on this in half-queue results later). In both TCPv3 and v4, the traffic shoots up when the interfering traffic stops since the accumulated traffic now gets a chance to be transmitted.

Figs. 9–14 show the same set of graphs when the interfering FTP traffic is halved to 4 Mb/s. This is still a fairly serious case of bandwidth shortage. The first thing to note from these graphs is a significant increase in TCP oscillations, especially for TCPv3. This is to be expected as TCP gets more chances to increase its window, only to be forced to cut back. We do not address this issue of TCP in this paper since it can be adequately addressed by well-known techniques such as delay based control [1] and slowly responsive congestion control algorithms [7]. Notice that in Fig. 9, there is little difference between TCPv3 and "TCPv4" cases since both are really TCPv3 for the FTP traffic.

It is seen from Figs. 10 and 11 that TCPv4 for DB traffic completely avoids any packet drops and retransmissions. As a result, the interfering traffic basically has no impact on DB1 and DB2 throughput for TCPv4. In contrast, TCPv3 still experiences significant losses, retransmissions and throughput irregularities. It is important to point out, however, that the loss prevention is primarily an effect of delay related control and should be observed with delay sensitive TCP versions such as Vegas or Fast. The relevant point to note from the perspective of this paper is that both connections continue to maintain 1:2 throughput ratio.

We repeated the runs with only 2 Mb/s interfering FTP traffic. The results are not shown for space reasons, but they were similar to 4 Mb/s case. In particular, TCPv3 still experienced losses and some sharp throughput dips (the total offered to inter-router link was slightly above 10 Mb/s). Needless to say, TCPv4 did just fine here. In fact, the only point of even doing this test was to check (a) whether the control goes in effect (it did), and (b) whether there are any problems in terms of staying on with the control during the congestion period (none), and (c) whether the control is released promptly and permanently when the congestion ends (it did).

C. *Half Queue Results*

Figs 15-17 show selected parameters with half-queue and 8 Mb/s interfering traffic. As expected, a shallower queue leads to much higher losses, but TCPv4 still shows a concentration of initial losses (when the control is taking effect) followed by much lower losses than for TCPv3. The losses do, however, result in sharp throughput dips (since we have not changed the rest of TCP). Still, the throughput is far smoother for TCPv4 than for TCPv3.

The most interesting result here is that TCPv4 does indeed cut down the two flows just enough to maintain approximately 2:1 ratio. Recall that the full queue case wasn't quite able to do that. In fact, we studied the *double queue* case as well (not shown here), and found that it primarily cuts down DB1 throughput. The conclusion is that large router queues have the effect of damping the effect of endpoint control mechanisms. Thus, there is interesting tradeoff: larger queues avoid losses but at the cost of less effective endpoint control, higher latencies, and higher cost. The level of communication traffic between nodes is also relevant here: at low traffic rates (often found with compute intensive grid applications), the relative impact of latencies is lower. In other words, the important aspect is the bandwidth delay product of the communication. As this number increases, the endpoint control becomes less and less precise.

Figs 18-20 show selected half-queue parameters under 4 Mb/s interfering traffic. The results here can almost be predicted from the cases considered above. The choppiness of the traffic increases because of less severe overload, however, as a result of a small router queue, the DB1 to DB2 throughput

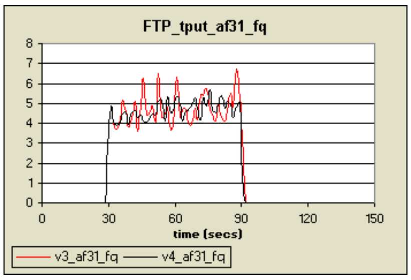

Fig. 3. FTP carried traffic: AF31, FullQ, 8 Mb/s FTP traffic

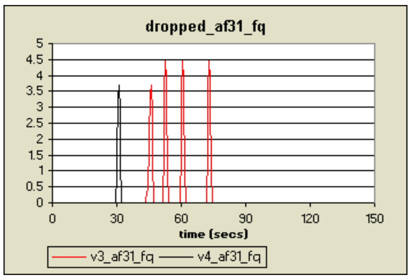

Fig. 4. IP Packet Losses: AF31, FullQ, 8 Mb/s FTP traffic

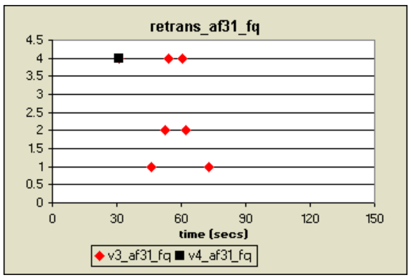

Fig. 5. Retransmissions for DB1: AF31, FullQ, 8 Mb/s FTP traffic

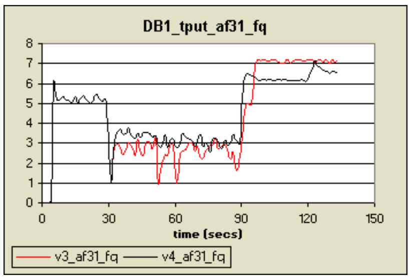

Fig. 6. DB1 carried traffic: AF31, FullQ, 8 Mb/s FTP traffic

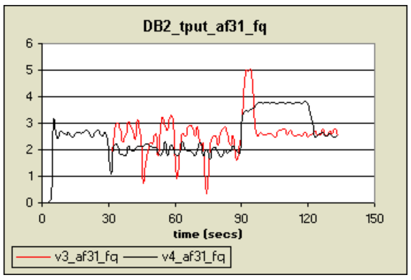

Fig. 7. DB2 carried traffic: AF31, FullQ, 8 Mb/s FTP traffic

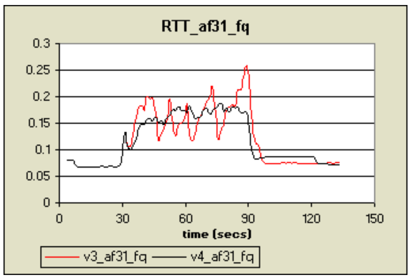

Fig. 8. TCP RTT for DB1: AF31, FullQ, 8 Mb/s FTP traffic

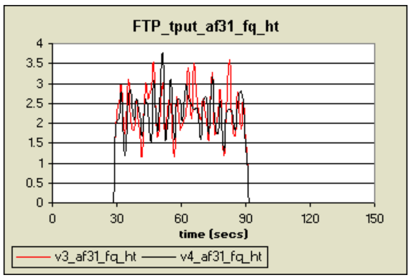

Fig. 9. FTP carried traffic: AF31, FullQ, 4 Mb/s FTP traffic

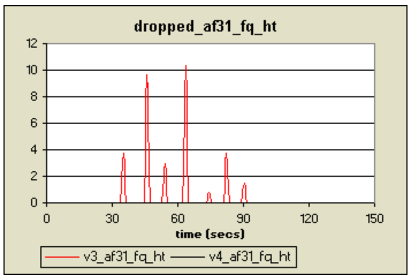

Fig. 10. IP Packet Losses: AF31, FullQ, 4 Mb/s FTP traffic

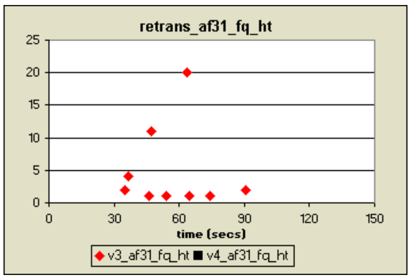

Fig. 11. Retransmissions for DB1: AF31, FullQ, 4 Mb/s FTP traffic

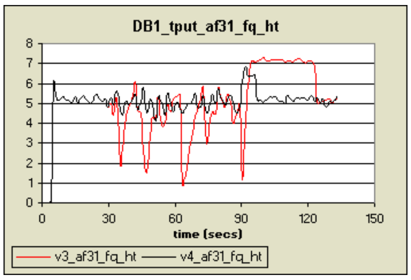

Fig. 12. DB1 carried traffic: AF31, FullQ, 4 Mb/s FTP traffic

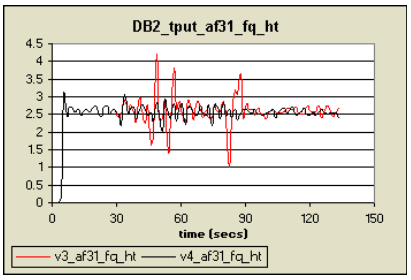

Fig. 13. DB2 carried traffic: AF31, FullQ, 4 Mb/s FTP traffic

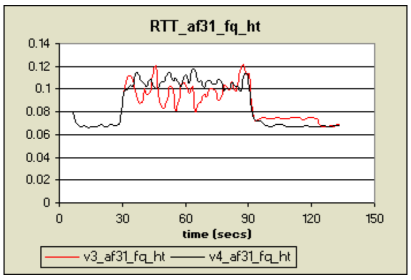

Fig. 14. TCP RTT for DB1: AF31, FullQ, 4 Mb/s FTP traffic

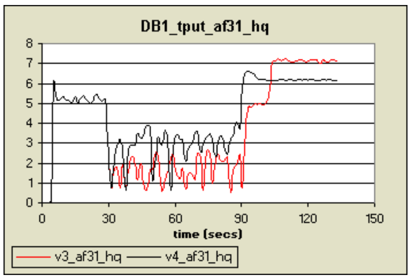

Fig. 15. DB1 carried traffic: AF31, HalfQ, 8 Mb/s FTP traffic

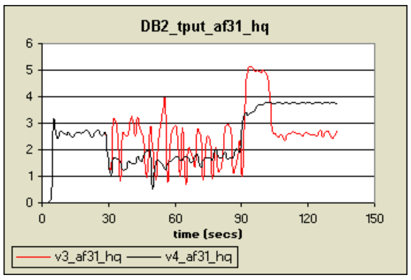

Fig. 16. DB2 carried traffic: AF31, HalfQ, 8 Mb/s FTP traffic

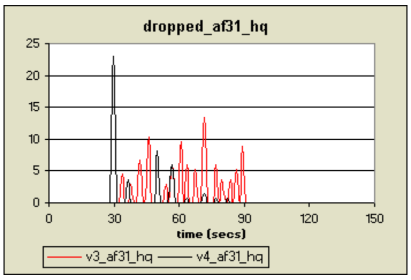

Fig. 17. IP Packet Losses: AF31, HalfQ, 8 Mb/s FTP traffic

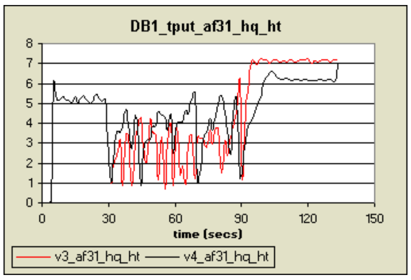

Fig. 18. DB1 carried traffic: AF31, HalfQ, 4 Mb/s FTP traffic

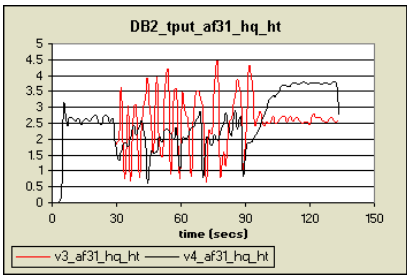

Fig. 19. DB2 carried traffic: AF31, HalfQ, 4 Mb/s FTP traffic

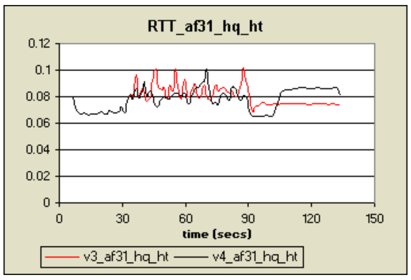

Fig. 20. TCP RTT for DB1: AF31, HalfQ, 4 Mb/s FTP traffic

is able to maintain the approximate 2:1 ratio. The losses (not shown) are still smaller for TCPv4 and concentrated initially. Consequently, the throughput is smoother for TCPv4 than for TCPv3.

It is worth focusing a bit on the behavior beyond time 90 secs when the interfering traffic goes away. It is evident from Figs 15, 16, 18, and 19, that TCPv4 recovers a bit more slowly than TCPv3 when the congestion ends. (This is related to the choice of η factor). As a result, there is a small period where the RTT falls to a normal level in TCPv4 before climbing up (see Fig 20). This is the "trap" that we talked about in the context of control removal. Also note that since the control stays on during the backlog period after the end of congestion, TCPv4 attempts to achieve the 2:1 throughput ratio even during this period. In contrast, TCPv3's behavior in this case is entirely dependent on how much backlog each connection accumulated during the congestion period.

D. Equal Priority Results

As stated earlier, by default we put FTP traffic at a higher priority (AF31) in order to allow it to inflict maximum damage to the DB flows of interest. In this section, we consider the situation where all 3 flows are at the same priority level. The equal priority case is perhaps more relevant when we consider non-streaming type of application traffic since no priority setups will usually be in place in practice.

With equal priority, the actual DSCP choice (and corresponding queue size and other parameters) determine the overall performance. Figs 21 -23 show selected results with 8 Mb/s interfering traffic and all flows given AF21 DSCP. Although one might expect that the interfering at a lower priority will cause fewer losses/retransmissions in the DB traffic, just the opposite happens. This can be seen from Fig 5 vs. 23 and is a result of more severe packet drops at the router by the RED mechanism due to smaller overall buffer size capacity. Consequently, the throughput behavior is less stable than in section IV-B but otherwise similar.

Figs 24 -26 show selected results with 4 Mb/s interfering traffic and all flows given AF21 DSCP. Again, the behavior is as expected: higher packet drops by RED results in less stable behavior than in section IV-B, but TCPv4 performs significantly better than TCPv3.

Finally, Figs 27 -29 and 30 -32 present equal priority results for the half-queue case, but with 4 Mb/sec and 2 Mb/sec FTP traffic respectively. As expected, the packet drops and retransmissions increase further due to shallower queue, so much so that even the 2 Mb/sec FTP traffic is no longer loss free. In turn, the behavior becomes much less stable. Still, TCPv4 behaves much less erratically than TCPv3. Furthermore, it is able to achieve the approximate 2:1 ratio in both cases. (This is not surprising since small router queue allows endpoint to have an effective control.)

Several other cases such as DB traffic using DSCP of AF31 and FTP using DSCP of AF11 were also used, but the space does not permit a detailed discussion. In particular, when all flows use AF31, TCPv4 is able to avoid losses completely, but TCPv3 still has a few losses.

To summarize, in all cases. the collective control outperforms the default TCP behavior not only in terms of providing approximate proportional fairness but also in terms of reducing retransmissions and losses. We expect these results to hold under circumstances not examined here; however, given an extremely large set of parameters relating to IP, QoS setup and TCP, it is difficult to exhaustively test the modifications.

V. Conclusions and Future Work

In this paper we studied the concept of coordinated bandwidth control in utility computing environments and showed in a simple setting how such a scheme can benefit applications. It was shown that the controls work very well and even reduce the packet losses significantly so long as the bandwidth delay product of the communications are not too large. In addition to the coordinated BW allocation, the most significant contribution of the paper is the autonomic mechanism to monitor traffic and set/remove controls. Although the scheme was tested with TCP, it applies equally well to SCTP or other reliable connection oriented protocols. Part of the further work on this topic is to show how the scheme can benefit more complex applications and its more through evaluation with a variety of TCP options. Another topic of further study is simultaneous use of TCPv4 for multiple interfering flows (e.g., for FTP & DB traffics in this study).

References

[1] C. Jin, D. X. Wei, and S. Low, "Fast TCP: Motivation, Architecture, Algorithms, Performance", Proc. of Infocom 2004.
[2] S. Jin, L. Guo, et. al., "A spectrum of TCP friendly window based congestion control algorithms", IEEE/ACM trans on networking, vol 11, no 3, June 2003.
[3] J. Martin, A. Nilsson and I. Rhee, "Delay based congestion avoidance in TCP", IEEE/ACM trans on networking, vol 11, no 3, pp356-369, June 2003.
[4] S.H. Low, F. Paganini, et. al., "Linear stability of TCP/RED and a scalable control", Computer Networks, vol 43, no 5, pp633-647, 2003.

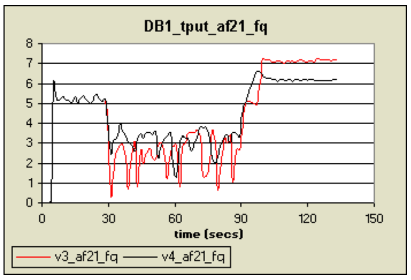

Fig. 21. DB1 carried traffic: AF21, FullQ, 8 Mb/s FTP traffic

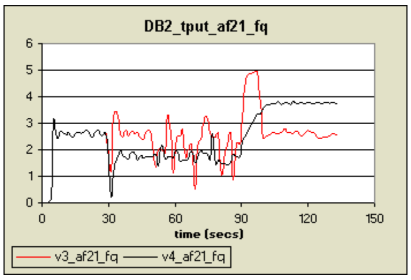

Fig. 22. DB2 carried traffic: AF21, FullQ, 8 Mb/s FTP traffic

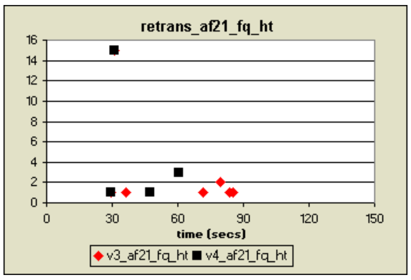

Fig. 23. Retransmissions for DB1: AF21, FullQ, 8 Mb/s FTP traffic

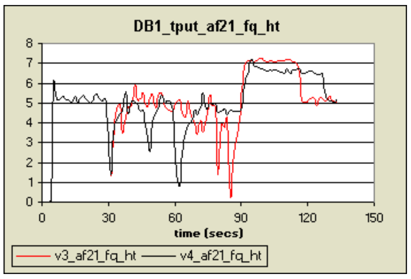

Fig. 24. DB1 carried traffic: AF21, FullQ, 4 Mb/s FTP traffic

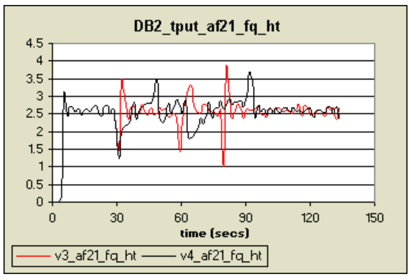

Fig. 25. DB2 carried traffic: AF21, FullQ, 4 Mb/s FTP traffic

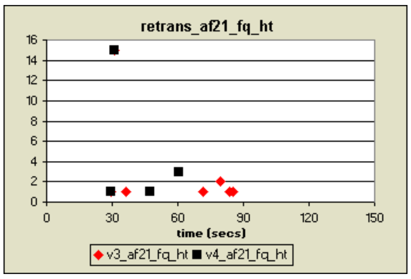

Fig. 26. Retransmissions for DB1: AF21, FullQ, 4 Mb/s FTP traffic

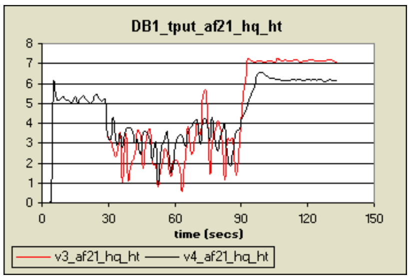

Fig. 27. DB1 carried traffic: AF21, HalfQ, 4 Mb/s FTP traffic

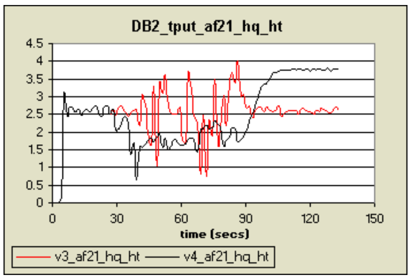

Fig. 28. DB2 carried traffic: AF21, HalfQ, 4 Mb/s FTP traffic

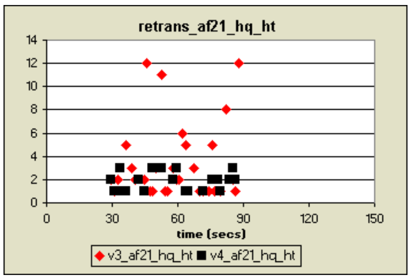

Fig. 29. Retransmissions for DB1: AF21, HalfQ, 4 Mb/s FTP traffic

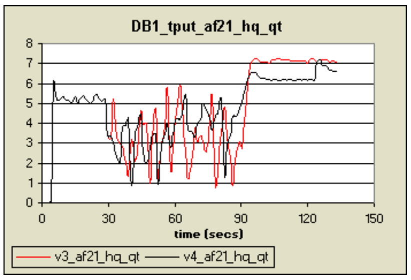

Fig. 30. DB1 carried traffic: AF21, HalfQ, 2 Mb/s FTP traffic

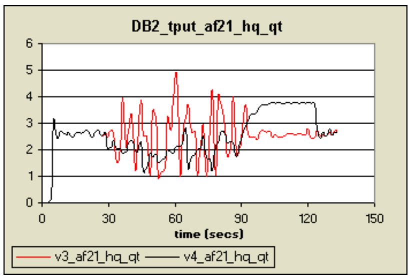

Fig. 31. DB2 carried traffic: AF21, HalfQ, 2 Mb/s FTP traffic

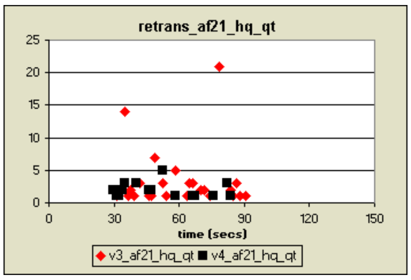

Fig. 32. Retransmissions for DB1: AF21, HalfQ, 2 Mb/s FTP traffic

[5] F.P. Kelly, A. Mullo and D. Tan, "Rate control for communication networks: shadow prices, proportional fairness and stability", Journal of OR society, vol 49, No 3, pp237-252, 1998.
[6] J. Crowcroft and P. Oechslin, "Differentiated end to end internet services using a weighted proportional fair sharing", ACM Sigcomm, vol 28, no 3, July 1998.
[7] D. Bansal, H. Balakrishna, et. al., "Dynamic behavior of slowly responsive congestion control algorithms", Proc. of SIGCOMM 2001.
[8] S. Cho and R. Bettati, "Collaborative Congestion Control in Parallel TCP Flows", Tech Report, Texas A&M Univ, 2004.
[9] S. Cho and R. Bettati, "Aggregated Aggressiveness Control on Groups of TCP Flows", Proc. of Networking 2005.