PLATO: Predictive Latency-Aware Total Ordering

Mahesh Balakrishnan, Ken Birman and Amar Phanishayee

{mahesh, ken, amar}@cs.cornell.edu

Department of Computer Science

Cornell University

Keywords: Datacenters, Distributed Systems, Group Communication, Multicast, Replication, Total Ordering.

Abstract

PLATO is a predictive total ordering protocol designed for low-latency multicast in datacenters. It predicts out-of-order arrival of multicast packets by observing their inter-arrival times, and delays packets before passing them up to the application only if it believes the packets to have arrived in the wrong order. We show through experimentation on real datacenter-style networks that the inter-arrival time of consecutive packet pairs is an excellent predictor of out-of-order delivery. We evaluate an implementation of PLATO on the Emulab testbed, and show that it drives down delivery latencies by more than a factor of 2 compared to the fixed-sequencer protocol.

1 Introduction

Total ordering is a fundamental problem in distributed systems - in simple terms, it refers to the task of ensuring that a set of nodes deliver incoming multicast messages in the same order. The total ordering problem is defined within the context of the *group communication* paradigm, where processes communicate with each other using multicast groups providing different message delivery semantics.

Total ordering protocols occupy a critical slot in the communication stack of modern commercial datacenters, allowing applications to distribute and replicate data and functionality with strong consistency guarantees. Made popular by online e-commerce websites, datacenters are increasingly the computing platform of choice for a wider range of applications, ranging from computational finance to mission-critical applications. Application domains traditionally centered around expensive, specialized hardware and software - such as air-traffic control or military command-and-control - have begun to migrate towards commodity datacenters, lured as much by the cheap running costs and easy maintainability of COTS components as by the possibilities of the scale-out paradigm - massive scalability and very high availability.

A growing class of datacenter applications is time-sensitive and requires low-latency delivery of multicast messages. In some cases, timeliness is directly related to real-world metrics - for instance, the inventory service of

an online bookseller has to reflect the latest item counts, to prevent losses caused by overselling. Other examples involve calculators running in financial datacenters, which require up-to-date information on stock quotes, and tracking applications on military datacenters dealing in location updates of targets. Such applications require the ability to spread data consistently and rapidly throughout a datacenter - mandating the need for a fast, low-latency total ordering protocol.

In this paper, we present PLATO, an optimistic total ordering protocol designed for time-critical datacenter applications. The key idea behind PLATO is to delay incoming data packets from the application only if there is a significant likelihood that they might be out of order - and to use a predictive scheme to determine that likelihood. PLATO allows applications to consume most incoming data packets within a few hundred microseconds of arrival, delaying packets to wait for ordering information only when it believes their arrival order to be inconsistent across the multicast group. In line with other optimistic ordering protocols [6, 7, 8], PLATO requires the application to have *rollback* capability, allowing delivered packets to be revoked when predictions are incorrect.

The predictor of out-of-order arrival used by PLATO is the inter-arrival time of consecutive packet pairs into user-space. Packet inter-arrival time is a simple yet powerful predictor of disorder in a datacenter setting - importantly, it is local information available at no extra cost or instrumentation at the receivers. To our knowledge, PLATO is the first predictive total ordering protocol - while there is at least one protocol that masks delay differences between receivers to achieve total order [8], we are not aware of any existing protocol that attempts to speculate on disorder on a per-packet basis.

The contributions of this paper are:

- We experimentally assess the causes and extent of out-of-order delivery on two datacenter-style switched networks - the Emulab testbed at Utah and a 252-node cluster at Cornell University.
- We propose the usage of inter-arrival times of consecutive packet pairs as a predictor of out-of-order delivery. We motivate this predictor by experimentally observing a high correlation between low inter-arrival times

and out-of-order delivery.

- We design and implement PLATO, a predictive total-ordering protocol that uses the above predictor to decide whether arriving packets are in order or not - and waits for extra ordering information only in the latter case.

- PLATO is evaluated on the Emulab testbed, and performs significantly better than the existing fixed-sequencer protocol, slashing delivery latency by more than a factor of 2 while incurring less than 1% rollbacks.

In Section 2, we articulate the requirements of a time-critical datacenter ordering protocol. In Section 3, we assess the extent and causes of out-of-order delivery on datacenter-style networks, and show that the inter-arrival time of packets can be an excellent predictor of disorder. Section 4 provides the design and implementation details of PLATO, and Section 5 is the evaluation of the implementation.

2 The profile of a datacenter total ordering protocol

A time-critical total ordering protocol is likely to co-exist on nodes with other protocols, competing for bandwidth and CPU cycles. A typical design for a datacenter application is shown in 1.(a), where several nodes host a replicated service; they are queried by other nodes via TCP/IP or some other unicast protocol, and updated using totally ordered multicast. For example, the replicated inventory service mentioned earlier would receive updates from the service responsible for processing buy transactions and be queried by services requiring up-to-date information on the availability of items.

The existence of other competing protocols and a time-critical, possibly CPU-intensive application running on the node emphasizes the need for a light-weight, low-overhead ordering mechanism. Datacenter workloads are likely to vary due to external factors - Christmas season for online stores and high-activity periods for stock calculators come to mind - and the ordering protocol should be capable of working well at different data rates, providing timely delivery at low and intermediate throughputs while being able to sustain bursts of high-throughput

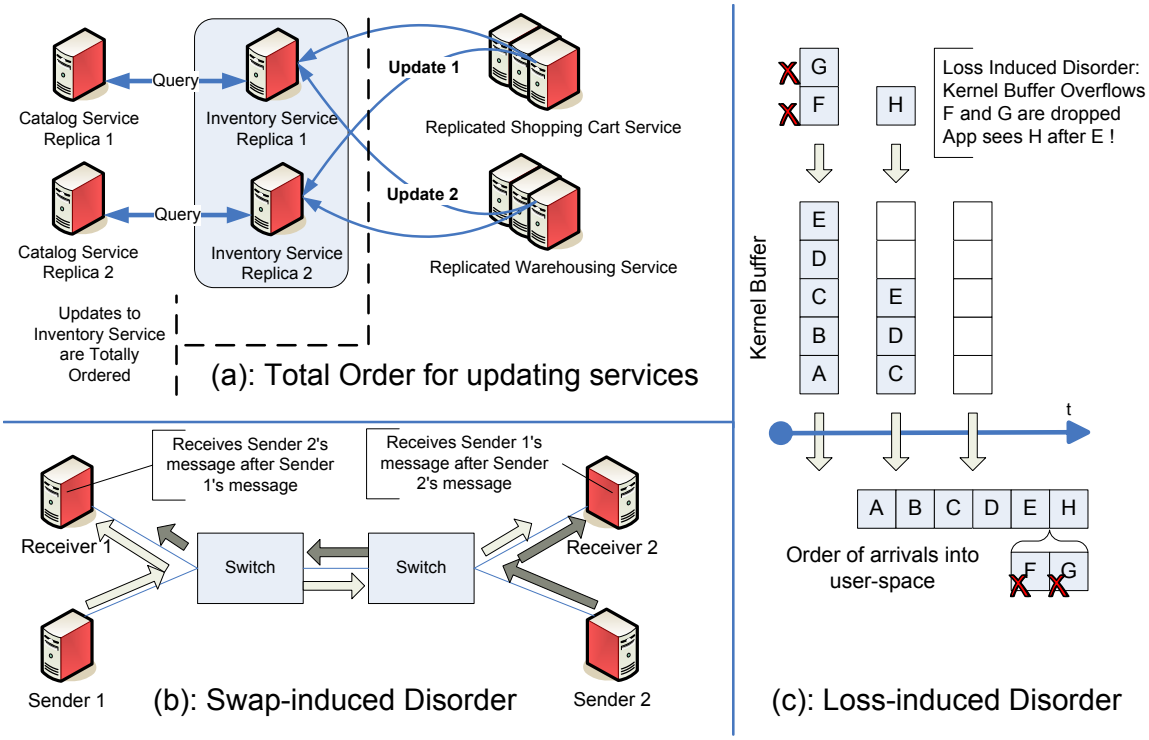

Figure 1.

traffic. A related goal is throughput and performance stability, typically achieved by inducing exclusively proactive overheads and avoiding costly reactions to failures that further destabilize the system. Additionally, datacenters and large clusters exhibit specific failure modes and performance trade-offs, and the ideal ordering protocol for such settings should be able to exploit the natural properties of the underlying hardware - while retaining the ability to work well on many different kinds of commodity hardware.

Of this wishlist of properties, we would like to underscore the importance of performance at low and rapidly varying data rates. We have argued elsewhere that the natural use of multicast in a datacenter gives rise to large numbers of groups with low individual data rates [1]. Imagine a replicated data store where fine-grained objects are cloned and cached on different nodes with high-level objectives such as fault-tolerance and data locality; if multicast is used to update objects, each node has to belong to as many groups as the number of objects it caches or replicates, resulting in large numbers of groups that overlap in chaotic patterns. Another example involves

financial calculators that use publish-subscribe libraries to subscribe to the latest prices for different equities, and hence belong to as many multicast groups as the equities they are interested in. The activity level within a single group can vary dramatically, even if the overall system throughput stays constant - the popularity of a single replicated object could experience sharp spikes, either independently or in correlation with other objects.

To summarize the properties mentioned thus far:

- The protocol should leverage the natural properties of the datacenter hardware,
- impose minimal and stable overheads,
- and crucially, work well at low per-group data rates that vary sharply over time and across groups.

3 Cluster Properties

It is a well-known fact that broadcasts on LANs arrive almost simultaneously at all receivers, and consequently the arrival of packets in different orders at different receivers is a very rare event. Multiple protocols have leveraged this property to provide optimistic delivery of broadcast messages to the application [6, 7]. In this paper, we extend this observation to IP Multicast [2] within datacenters.

Datacenters are typically heterogenous agglomerations of smaller homogenous clusters, interconnected by high-capacity switches. Intuitively, out-of-order delivery in switched networks occurs in two forms: *swaps* and *packet loss*. Swaps occur due to disparities in the distances between senders and receivers. Consider the simple case of two senders and two receivers illustrated in Figure 1.(b), where one sender is very close to one receiver and relatively far from the other one, and the other sender is placed close to the second receiver and far away from the first one. Nearly simultaneous multicasts from the two senders will arrive at different orders at the two receivers.

Packet loss in a datacenter almost never occurs within the networking fabric; more commonly, it is the end-host kernel that gets overwhelmed by the rate of incoming traffic and drops packets [1]. Figure 1.(c) illustrates how kernel buffer overflows trigger out-of-order delivery - the receiver delivers the packets immediately before and

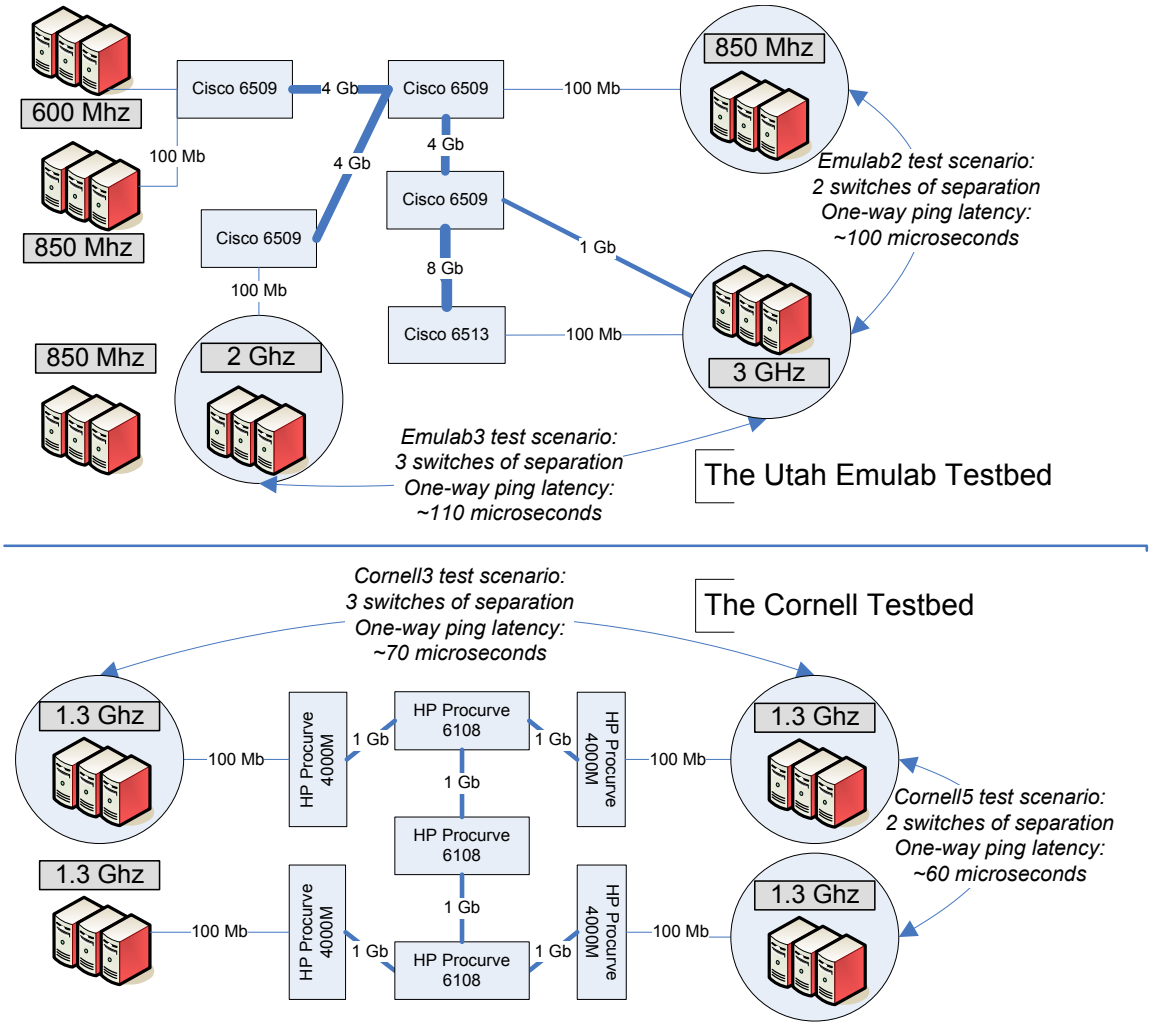

Figure 2. Clusters

after the loss burst in consecutive order.

3.1 Experiments

We ran simple experiments on two datacenter-style switched networks to evaluate the extent of out-of-order delivery of multicast messages. The first of these is the Emulab testbed at Utah [10], which is a heterogenous collection of several smaller clusters connected by Cisco 6500 series switches; Figure 2 shows the topology of the testbed (redrawn from information on www.emulab.net) - inter-node one-way ping latencies range from 100 to 300 microseconds, depending on the location of the nodes and how loaded the network is. The second network

is a homogenous rack-style cluster of 252 1.3 Ghz nodes at Cornell University, comprising of 14 racks of 18 blade-servers each, interconnected via a 3-level hierarchy with HP Procurve 4000M switches at the leaves and HP Procurve 6108 switches in the interior - the network diameter is around 60-100 microseconds.

Our experiment involved placing a receiver and a sender each on two different parts of the network separated by multiple switches, and multicasting data at different rates to measure the frequency of out-of-order deliveries. We ran this experiment in four scenarios - $Cornell3$ and $Cornell5$: the Cornell cluster, with the 1.3 Ghz sender-receiver pairs separated by three switches and five switches, respectively, $Emulab3$: the Emulab testbed, where one sender-receiver pair consists of 3 Ghz nodes and the other sender-receiver pair is made up of 2 Ghz nodes three switches away, and $Emulab2$: the Emulab testbed, where one pair consists of 3 Ghz nodes and the other of 850 Mhz nodes two switches away. Figure 2 outlines the placement of these four scenarios.

Figure 3 shows the percentage of swaps and losses in these four scenarios, as we increase the multicast sending rate in the group. We measure simple swaps by comparing receiver logs after the experiment and locating consecutive packets which are delivered in inverted orders at the two receivers. As expected, the higher the rate of multicasts, the higher the probability of a swap occurring - for the Cornell cluster 5-switch scenario, the percentage of consecutive packet pairs which are swaps rises from 1% at 800 packets per second to 4% at 4000 packets per second. For the Emulab 3-switch scenario, the percentage of swaps rises from 0.7% at 800 packets a second to around 3.2% at 4000 packets per second. In these graphs, we do not show the frequency of more complex swap events, where a sequence of packets is swapped with another sequence - we observed a very small percentage $(< 0.0001\%)$ of these on the Cornell cluster, and none of them on the Emulab testbed.

Figure 3 also shows that packet loss increases with receive throughput, albeit less smoothly - the Cornell 5-switch scenario loses more packets at 2000 packets a second than at 2400 packets a second, and the Emulab 3-switch scenario exhibits more loss at 3200 packets per second than at 3600 packets per second. Our hypothesis for this uneven increase in packet loss is that the inter-arrival time of packets interacts with the OS thread scheduling mechanisms in complex ways - at intermediate rates, the receive thread is occasionally context-switched out and

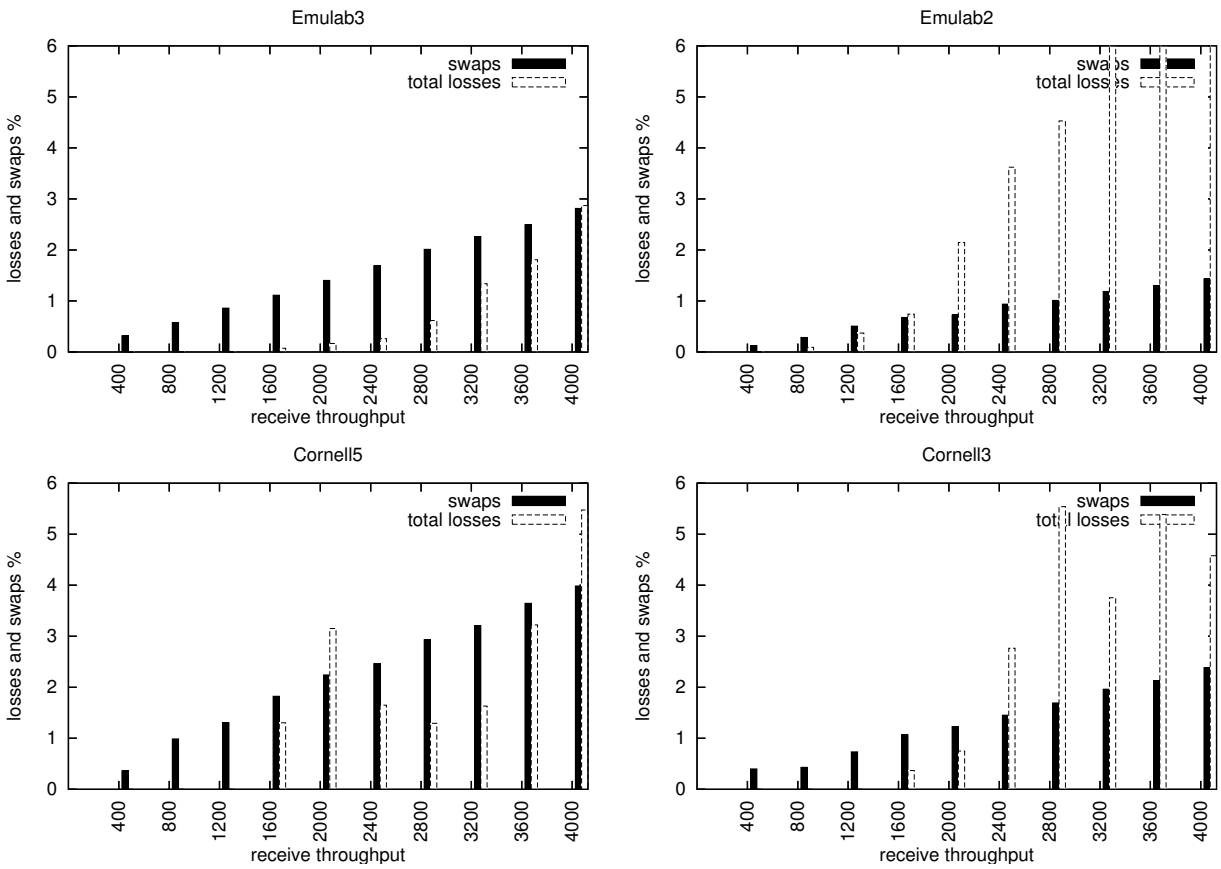

Figure 3. Disorder Characterization

loses packets while it's not running, whereas at very high rates the receive thread is continuously dequeueing packets off the socket and hence is rarely context-switched out.

With this experiment, we established that out-of-order delivery does occur in switched datacenter-style networks. Next, we explore the feasibility of using the inter-arrival time of consecutive packets into user-space as a predictor of both swaps and packet loss. Since swaps occur when multicasts are nearly simultaneous, it is natural to hypothesize that a swap would involve two packet arrivals that are very close to each other - in this case, we expect the arrival times of packets into user-space to reflect the actual timing of the multicasts. Since packet loss occurs when kernel buffers overflow, we'd expect to observe a sequence of very low user-space inter-arrival times immediately prior to the loss burst, as the receive thread rapidly empties packets from the full kernel buffer. Recall

that we explained these scenarios in Figure 1.

To validate our hypotheses, we examined distributions of inter-arrival times for consecutive packet pairs. We are interesting in two metrics of the distributions:

- The time representing the 95th percentile of inter-arrival times of swapped packet pairs, and
- the percentage of all consecutive packet pairs - swapped or not - whose inter-arrival times fall within this limit.

Figure 4 shows this data in six settings: the top three graphs are for different throughputs, in the *Cornell3* scenario, and the bottom three are for a single throughput setting of 1200 packets per second, for the *Cornell5*, *Emulab2* and *Emulab3* scenarios. The top half of each graph shows the histogram for the inter-arrival time intervals for swapped packet pairs, and the vertical line in each graph is the 95th percentile of these intervals. The bottom half shows the inter-arrival time for all packet pairs, and as the vertical line continues down into this half, it indicates the percentage of inter-arrival times of all packet pairs that lie within it. The two metrics mentioned are stated on top of each distribution graph.

Why are we interested in knowing the percentage of all packet pairs that fall within the 95th percentile of swapped pair inter-arrival times? In the *Cornell3* 400 packet graph (top, left), 95% of all swaps and 14% of all packets have inter-arrival times of less than 128 microseconds. Hence, if we use an inter-arrival threshold of 128 microseconds to detect swaps - by raising a ‘red flag’ (we will elaborate later on what exactly this entails) whenever two packets arrive within that threshold time of each other - we would end up catching 95% of all swaps, and suspect 14% of all packet pairs of being swaps.

Figure 5 shows how the two metrics mentioned above vary with throughput, for the four different scenarios; this gives us a better understanding of how data rate affects the quality of prediction that can be obtained by observing the inter-arrival times. In the graph, the 95th percentile of inter-arrival times of swaps stays almost constant for all the scenarios - however, the percentage of all packet pairs that fall inside it goes up significantly

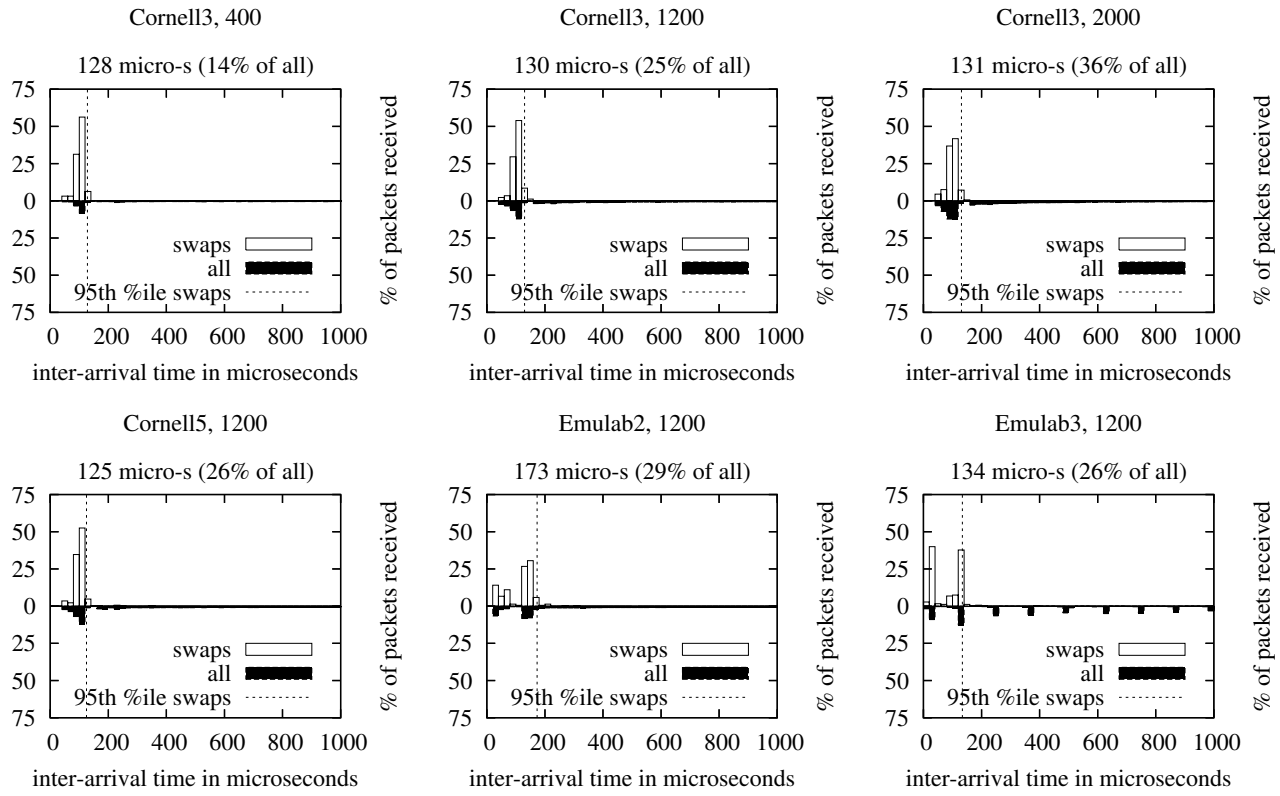

Figure 4. Histograms of Packet Inter-arrival Times

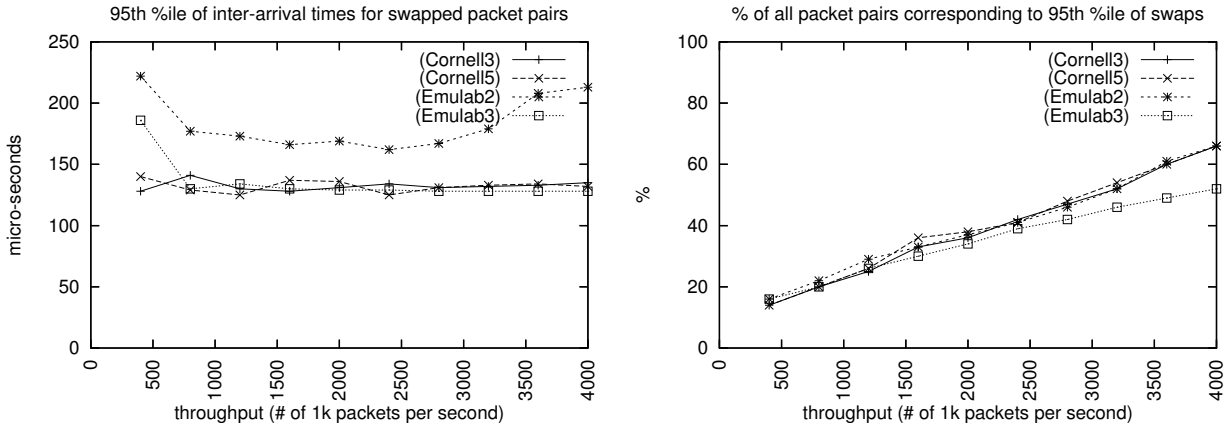

Figure 5. Variation of Swap metrics with throughput

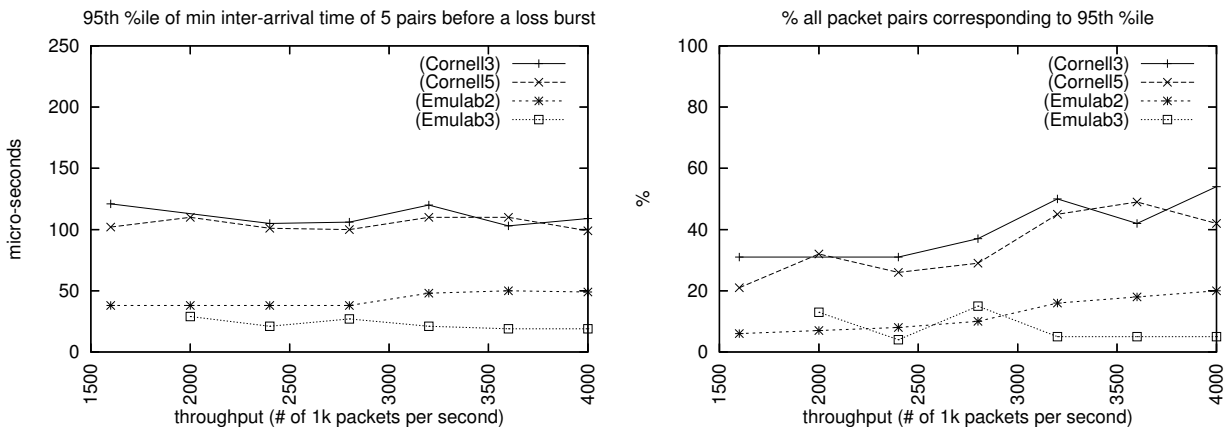

Figure 6. Variation of Loss metrics with throughput

as throughput rises. To continue using the metaphor of a 'red flag', a fixed threshold would catch all swaps, irrespective of throughput; however, it would also suspect a much higher percentage of all packet pairs of being swaps.

Next, we perform a very similar analysis of losses. Here, we measure the minimum inter-arrival time of the last five packet pairs immediately preceding a loss burst event. The rationale here is that the 'red flag' in this case gets raised when we observe a sequence of low inter-arrival times. Figure 6 shows this data; note that for certain throughput-scenario pairings we did not observe enough loss to compute a percentile, and we have not plotted these on the graph.

The data presented thus far leads us to formulate the following heuristic for detecting disorder, parameterized by a threshold Δ - *if the inter-arrival time between two packets at a receiver is less than Δ, it is reasonably probable that the packets have arrived in different order at other receivers; if the time is greater than Δ, it is highly improbable that the packets are delivered in different order at other receivers.*

This heuristic fits swap-induced disorders precisely - and, in our particular implementation of it, also suffices to catch loss-induced disorder; this will become clear once we describe our design.

4 The Design of a Predictive Ordering Protocol

To embed the heuristic presented above into a practical protocol, we need to examine the design space of total ordering algorithms. Defago, *et al.* [3] provide an analytical comparison of total ordering protocols, dividing them into five broad categories - *fixed-sequencer, moving-sequencer, privilege-based, communication history, and destinations agreement* - and conclude that for the non-uniform version of total ordering (where the delivery order at failed nodes does not matter), fixed-sequencer has the least latency and the second-highest throughput. The paper states that the moving-sequencer algorithm has slightly higher throughput than fixed-sequencer, but is more complicated to implement. Accordingly, we focus on the fixed-sequencer algorithm, both as a performance benchmark to compare against and as a starting point for our own design.

In the fixed-sequencer algorithm, a single receiver - the *sequencer* - periodically multicasts *sequencing messages*, establishing the correct order of delivery for the rest of the group. While most theoretical discussions of fixed-sequencer present the algorithm as sending out a single sequencing message for every received data message, in practice the sequencer can send out an ordered list of multiple received data messages in every sequencing message, allowing it to tune the overhead imposed by sequencing. If the sequencer sends one sequencing message for every k received data messages, one out of every $k+1$ messages in the group is pure sequencing overhead; we call k the *trade-off* parameter.

Armed with this basic understanding of the operation of the fixed-sequencer protocol, it becomes apparent that in most cases, messages are delayed unnecessarily while the receiver waits for ordering information from the sequencer. An ideal total ordering algorithm would delay only those packets which are received in different orders at different receivers, and deliver all other messages immediately to the application. This observation leads us towards a redesign of the fixed-sequencer protocol, where receivers wait for sequencing messages before delivering packets to the application only if they suspect them of arriving in the wrong order. What is needed is a decision mechanism that can be applied on a per-packet basis to selectively wait for sequencing information, and

this is precisely the heuristic described in the previous section.

For a predictive mechanism to be a viable option, the application should provide some form of *rollback* capability to the protocol. We assume that the application provides the protocol with hooks that allow packets to be *revoked* once they are delivered, causing a rollback if the application has already consumed the packet. Since application rollbacks are likely to be very expensive, we aim at having a very low percentage of them - typically, one out of every thousand packets. Note that a packet revocation will only trigger an application rollback if the application has already consumed the packet.

4.1 PLATO: Design and Implementation

The basic idea behind PLATO is extremely simple: If two consecutive packets arrive within Δ of each other, they are suspected of arriving in incorrect order and further information is awaited from the sequencer node before they are delivered to the application.

A trivial implementation of this idea would involve delaying packets for Δ time and delivering them to the application if no other packet arrives within that time. However, Δ is likely to be in the tens or hundreds of microseconds, making an efficient implementation of this algorithm difficult if not impossible on commodity platforms - context switching granularity is typically in the milliseconds, and varies greatly over time and across hardware.

As a result, PLATO does not delay packets before passing them up to the application - instead, it tags each packet with a timestamp T_m before which it should not be used by the application, equal to Δ microseconds after its arrival time. Instead of sleeping for Δ microseconds and then waking up and delivering the packet to the application if no other packet arrives within that time-span, it just delivers the packet to the application and resumes listening on the socket. If another packet arrives within Δ microseconds, we revoke the last packet from the application instantly - since we are within the Δ envelope, we can be sure that the packet has not been consumed by the application, and hence the revocation does not trigger a potentially expensive application-level

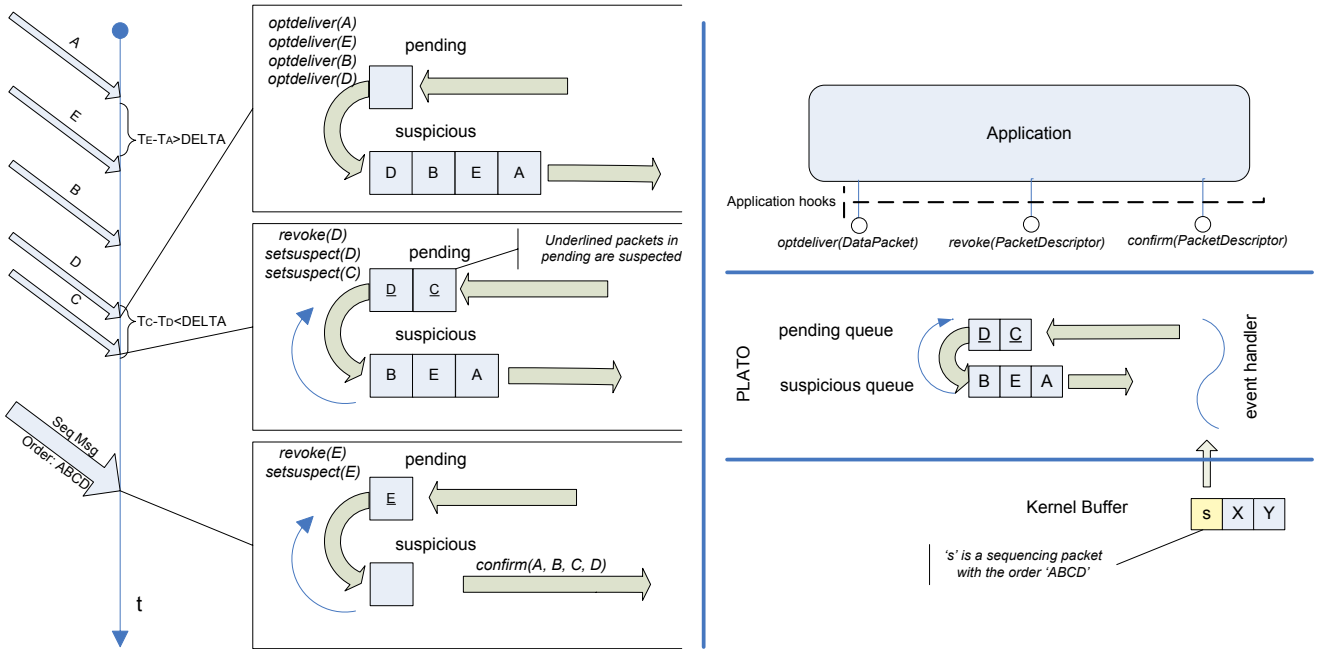

Figure 7. The PLATO Pipeline

rollback. An important metric for the protocol, then, is the frequency with which a revocation of a packet occurs after the corresponding time T_m has passed.

PLATO has three hooks into the application - *optdeliver*, which takes a data packet as a parameter and is called to optimistically deliver packets which may later be revoked; *revoke*, which takes a packet descriptor as a parameter and is called to revoke packets previously *optdeliver*-ed to the application, and *confirm*, which is called with the packet descriptor of a previously *optdeliver*-ed packet when the final ordering of that packet is known.

As shown in Figure 7, PLATO processes packets through a simple pipeline consisting of two queues - a *pending* queue, which consists of packets being conservatively withheld from the application, and a *suspicious* queue, which consists of packets already sent up to the application for which sequencing information has not yet arrived, and which can consequently be revoked from the application. Packets in the *pending* queue are marked *suspected* of being out-of-order and will not be delivered to application until sequencing information arrives for them; or, they are not *suspected* but are stuck in the queue behind one or more *suspected* packets. If no out-of-order arrivals occur, data packets travel through the *pending* queue to the *suspicious* queue, and then onto the application; if they

do occur, the arrival of sequencing information can cause packets to be transplanted from the middle of one queue to the other, or to the application, violating queue FIFO order.

In addition, PLATO maintains a *sequencing* queue, which buffers sequencing information - this queue comes into play only if we receive sequencing information for a data packet which we have not yet received, and hence have to queue up all subsequent sequencing information until that data packet arrives and can be delivered to the application.

We now describe PLATO in terms of two simple events - the arrival of a data packet, and the arrival of a sequencing packet. When a data packet is received, PLATO tags it immediately with the arrival time. If ordering information for the data packet has already arrived from the sequencer, the packet is *optdeliver*-ed to the application and immediately *confirm*-ed, with no further processing. If not, the arrival time is compared with the previous data packet's arrival time, and the difference checked against Δ.

If the difference is less than Δ, the packet is tagged as *suspected*, and added to the *pending* list. Now we need to locate the previous data packet in the PLATO pipeline and prevent it from being used by the application. There are three possibilities -

1. It is in the *pending* queue and has not been *optdeliver*-ed to the application, in which case we can tag it as *suspected*.

2. It is the last packet in the *suspicious* queue and has been *optdeliver*-ed to the application, in which case we *revoke* it from the application and move it from the tail of the *suspicious* queue back to the head of the *pending* queue. Note that it is necessarily the last packet in the *suspicious* queue and cannot be in the middle, since it was the last packet to be received.

3. It is in neither the *suspicious* nor the *pending* queues, in which case it has already been sequenced and *confirm*-ed to the application. Nothing more has to be done in this case.

If the difference is more than Δ, the packet's fate depends on the contents of the *pending* queue. If the *pending*

queue is non-empty - i.e, there are packets ahead of the current packet which are tagged *suspected* and are awaiting ordering information - then we need to add this packet to the end of the *pending* queue. If the *pending* queue is empty, then we can *optdeliver* the packet and add it to the end of the *suspicious* queue.

The second part of the protocol concerns its behavior when a sequencing packet is received. In its practical implementation, a sequencing packet contains a list of data packet descriptors - sorted by the order of arrival at the sequencer node. We iterate over this list of descriptors, and for each of them we locate the corresponding data packet within the PLATO pipeline (if we can't locate it in the pipeline, we buffer the descriptor - and all descriptors following it in this and subsequent sequencing packets - in the *sequencing* queue until we receive the data packet). Once we locate the data packet, we perform one of the following actions:

1. If the packet is in the *pending* queue, we remove it and *optdeliver* and *confirm* it to the application. We also dequeue all the packets from the *suspicious* queue, *revoke* them from the application, tag them as *suspected* and move them back to the head of the *pending* queue; these are packets incorrectly delivered to the application ahead of the currently sequenced packet.

2. If the packet is in the *suspicious* queue, we remove it and *confirm* it to the application. In this case, we have to dequeue all packets ahead of it in the *suspicious* queue, *revoke* them, tag them as *suspected* and move them back to the head of the *pending* queue.

4.2 Implementation and other Details

PLATO is implemented as an event-driven system with two threads, one running at high priority dequeueing packets off the socket and pushing them into the event queue, and the other servicing the queue and processing events. The implementation is written in Java - for our experiments, we use Java's System.nanoTime() method for determining the current system time at microsecond precision; this may not be universally portable, but is implemented on most modern platforms.

PLATO runs a link reliability layer that uses sender-based sequencing and negative acknowledgments to request unicast retransmissions of lost packets. Node failure is orthogonal to the protocol as it is presented, and any scheme that works to handle such faults in fixed-sequencer protocols should work equally well here. Also, while we have presented PLATO as a modification of fixed-sequencer, we could equally well have modified a moving-sequencer algorithm with similar results.

5 Performance Evaluation

To evaluate PLATO, we ran it in the Emulab 3-switch setting - recall that this involves a 3 Ghz sender-receiver pair and a 2 Ghz sender-receiver pair on the Utah Emulab testbed, with three switches separating them. Our first experiment was aimed at observing the impact of the Δ parameter on the performance of the protocol. Figure 8 plots delivery latency against the left y-axis as we run PLATO at increasing values of Δ. The left sub-graph shows performance at 400 packets per second, and the right sub-graph at 1600 packets per second, at two different values of k - the *trade-off* parameter (from Section 4). The horizontal fixed lines in these graphs show the performance of fixed-sequencer ordering for the same *trade-off* parameter, and consequently for the same overhead. The bars at the bottom of these graphs - plotted against the right y-axis - show the fraction of packets that are rolled back. We see from these graphs that the higher the value of Δ, the higher the delivery latency and lower the fraction of rollbacks - also, PLATO always out-performs fixed-sequencer for the corresponding value of k.

Next, we examine PLATO's performance in the presence of changing throughput levels. In Figure 9, we start out with 2 senders sending 200 packets/second each, and add 2 more senders 20 seconds into the experiment sending at 750 packets/second each for a total of 10 seconds - hence, the data rate in the group jumps from 400 packets per second to 1900 packets per second between time $t = 20$ and $t = 30$. On the left y-axis of these graphs, we plot 1-second moving averages every 10 milliseconds, of the delivery latencies achieved by PLATO and fixed-sequencer. As we can see from the graphs, PLATO's delivery latency remains constant throughout the experiment, whereas fixed-sequencer's delivery latency varies drastically with the data rate. The bars at the bottom of the

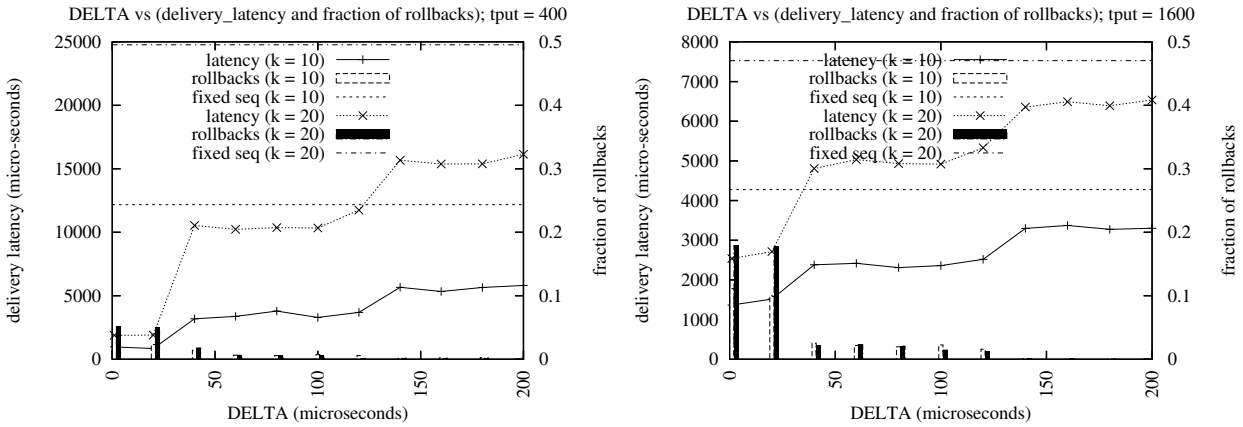

Figure 8. Δ vs Delivery Latency: 400 packets/second (left) and 1600 packets/second (right)

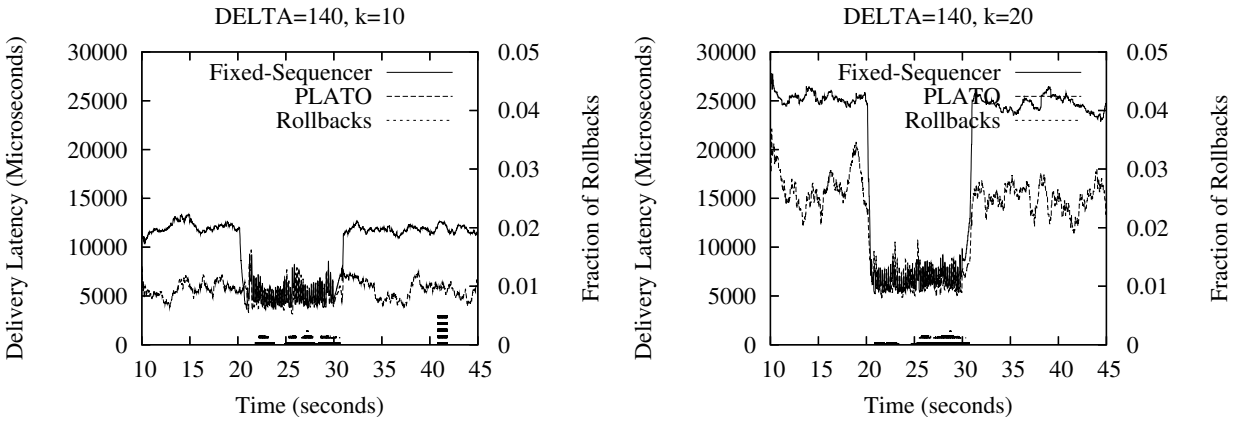

Figure 9. Traffic spikes up from 200 packets/sec to 1900 packets/sec between 20 and 30 seconds.

graphs is a 1-second moving fraction of rollbacks, plotted against the right y-axis. Note that for a higher value of k, delivery latencies are much higher for both protocols - since the latency to receiving ordering information from the sequencer node is higher.

Figure 10 shows how delivery latency and rollback fraction are affected by the data throughput, for a particular value of Δ. As the throughput goes up, the latency to receiving a sequencing packet goes down for a particular value of the *trade-off* parameter k, and consequently delivery latency drops. There is no real trend for rollbacks at this particular setting of Δ - all the values are within a tenth of a percent; however, for lower values of Δ we observe the fraction of rollbacks going up with throughput.

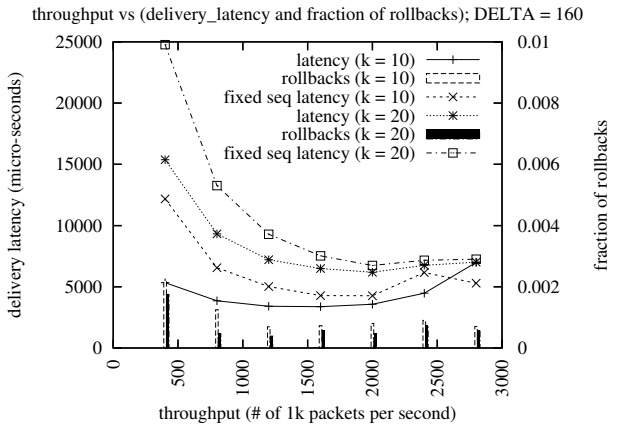

Figure 10. Throughput vs Delivery Latency

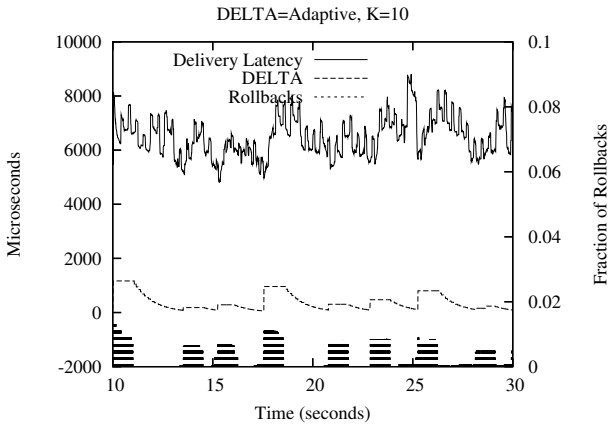

Figure 11. Setting Δ Adaptively

In Figure 11, we replace the static Δ parameterization of PLATO by a simple adaptive scheme. We multiply the current value of Δ by 1.5 when a rollback occurs and the current 1-second moving fraction of rollbacks is more than 0.01. Conversely, we multiply PLATO by .9 every 100 milliseconds or 1000 packets, whichever occurs first, if the moving fraction of rollbacks is less than 0.01. In Figure 11 we show that this simple mechanism gives good performance - for comparison, this setting is similar to the $k = 10$ scenario in Figure 9, during the traffic spike.

6 Related Work

A plethora of total ordering protocols exists in literature - we would like to point the reader to [4], which offers an excellent and thorough survey of this body of work, along with very useful categorizations. The particular subclass of ordering protocols that our work is closest to are the optimistic algorithms [6, 7, 9].

Sousa, et al. propose a solution for WANs where receivers observe network distances and delay packets appropriately to achieve a total ordering [8]. While this work is close in spirit to our own, it targets a completely different networking environment and uses a technique that works very well in the wide-area but may not be quite as useful in switched networks.

7 Conclusion

Low-latency data replication is a fundamental need for an emerging class of datacenter applications. PLATO is a total-ordering protocol designed for such settings - it targets the traffic patterns commonly observed in these applications and exploits the characteristics of the underlying hardware. We experimentally show that out-of-order delivery occurs to a reasonable degree on switched datacenter-style networks, and that the inter-arrival time of consecutive packet pairs is a powerful predictor of disorder.

8 Acknowledgments

We would like to thank Danny Dolev for his many valuable comments during the inception of this paper, Saikat Guha for a key discussion of protocol implementation, Mike Hibler for his quick responses on the Emulab testbed topology, and Art Munson and Einar Vollset for their feedback.

References

[1] M. Balakrishnan and K. Birman. Reliable multicast for time-critical systems. In *To Appear in Proceedings of the 1st Workshop on Applied Software Reliability (WASR 2006)*, 2006.

[2] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[3] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, December 2003.

[4] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[5] R. Guerraoui, R. Levy, B. Pochon, and V. Quma. High throughput uniform total order broadcast protocol for cluster environments. In *DSN*, June 2006.

[6] B. Kemme, G. Alonso, F. Pedone, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, 1999.

[7] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *DISC*, pages 318–332, 1998.

[8] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 190, Washington, DC, USA, 2002. IEEE Computer Society.

[9] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *SRDS*, pages 92–101, 2002.

[10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.