Causal Ordering in Deterministic Overlay Networks

Roy Friedman
Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000
Israel
roy@cs.technion.ac.il

Shiri Manor*
Intel Israel LTD
MATAM
Haifa 31015
Israel
shiri.manor@intel.com

April 18, 2004

Abstract

This paper *formally* proves that constrained flooding along the logical links of a deterministic overlay network preserves causal ordering, yet does not induce any control information overhead. The paper also analyzes the performance of such a mechanism when implemented over a hypercube overlay, and compares it to running a timestamp based protocol over a flat architecture. The hypercube overlay was chosen since it offers low fan-in/fan-out degree and is therefore highly scalable, it is fault-tolerant, and has small diameter.

Keywords: Distributed Systems, Overlay Networks, Causal Ordering, Scalability.

*The work was done while this co-author was a graduate student in the Department of Computer Science at the Technion

1 Introduction

Causal ordering ensures that messages are delivered in an order that matches the cause and effect relation, also known as the *happens before* relation, between their corresponding send events. This abstraction is useful for many applications. For example, in an internet mailing list or chat application, whenever a posting B is sent in response for another posting A, it is desirable that all users view A before B. Similarly, several works have shown how to achieve data-base consistency based on causal ordering, and that it is possible to obtain consistent snapshots of a system in which all messages are delivered in causal ordering without any additional costs [4, 13, 34, 36].

Causal ordering is also one of the strongest ordering requirements that can be guaranteed in a fully asynchronous distributed system prone to failures. In contrast, for example, total ordering cannot be obtained in such environments due to the FLP impossibility result [18]. However, direct implementations of causal ordering require piggybacking $O(n)$ control information and introduce delivery latencies when messages are received from the network in the wrong order. These overheads are reasonable for small systems, but become prohibitively expensive in large scale environments.

At the same time, application level *overlay networks* are becoming common practice in large scale dissemination applications. Specifically, with overlay networks, a logical topology is superimposed on the machines that participate in the system, and data is disseminated along the logical arcs of the overlay topology. Of course, the match between the logical topology of the overlay network and the physical underlying network can impact the performance of the system. However, one of the key features for scalability is limiting the fan-in and fan-out degrees of all nodes in the overlay network. (Note that the overlay topology is between end hosts and not IP routers.)

A fair amount of work has been published on various overlay topologies, some of which are deterministic, e.g., [6, 7, 19, 21, 27, 29, 32, 38, 42], and some of which are random, e.g., [10, 28, 41]. In this paper we formally prove that constrained flooding along the logical arcs of a deterministic overlay network preserve causal ordering at no additional cost.

We then concentrate on a hypercube topology. Hypercubes are attractive since they present good tradeoffs between the various system parameters. Namely, assuming n nodes in the system, the degree of each node is $O(n)$, the diameter is $O(n)$, and the number of node disjoint paths between each pair of nodes is also $O(n)$. The multiple node disjoint paths property implies fault tolerance. The small diameter results in short dissemination latencies. Finally, the small degree is important for the scalability of the approach. While hypercubes are defined only when the number of nodes is a power of two, we have shown in earlier work a construction for incomplete hypercubes that maintain all their nice properties [19, 30]. In this work, we conduct simulations of using a hypercube overlay and compare it to a direct implementation of causal ordering on a flat architecture. The results indicate that for small messages, the overlay based scheme is much more efficient despite its message redundancy. Short messages are common in applications like stock quotes, chat rooms, and many control applications.

We would like to emphasize that we do not claim to invent constrained flooding. Rather, our contribution is *formally* proving that it preserves causal ordering without additional control information, and analyzing its performance with respect to one appealing topology.

2 Definitions and Model

We assume a distributed system consisting of n processes connected by an interconnection network and communication by exchanging messages over the network. We do not make any assumption about the network latency, or the synchrony of processes. Each node includes an *application module* that runs the user's code, a *networking module* that is responsible for interacting with the network, and a *delivery mechanism* that is placed between them. The goal of the delivery mechanism is to extend the network guarantees and provide stronger semantics for the application module. For the sake of simplicity, in this work we assume that the network delivers messages reliably in FIFO order, and the goal of the delivery mechanism is to ensure causal delivery, as defined below. Clearly, it is possible to assume weaker network guarantees, and a structured delivery mechanism whose lower layers ensure reliable FIFO ordering, and the top layer providing causal ordering, as done in modern group communication toolkits like Horus [40], Ensemble [22], and JavaGroups [1].

We adopt the typical approach to modeling processes as deterministic automata. Processes may incur message receive events, and in response, perform some computation and generate zero or more send events. The send events create messages that the network has to deliver to the corresponding recipients. In this work we assume that each message is sent as a broadcast to all nodes in the system. We define a *history* of a process to be the sequence of events it incurs and generates, and an *execution* to be a collection of histories, one for each process, in which for every message receive event there is also a corresponding message send event. We also use the following notation thoughout this work: $send_{p_i}(m)$ denotes the send event in which the application module of process p_i issues a message m to be sent by the delivery mechanism; $delp_i(m)$ denotes the delivery event in which the delivery mechanism passes message m to the application module; $net\_send_{p_i}(m)$ is the send event in which the delivery mechanism passes m to the network; $recv_{p_i}(m)$ is the receive event in which the delivery mechanism of p_i obtains m from the network.

Definition 2.1 (happens before relation): *An event $a \rightarrow b$ iff*

1. *a and b are two events of the same process and a occurs before b.*
2. *a is a* `message-send` *event $send_{p_i}(m)$ and b is the corresponding* `message-deliver` *event $del_{p_j}(m)$, for some process p_j.*
3. *There exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.*

We assume that the *happens-before* relation does not contain cycles (it is a partial order), which means that message delays are grater than zero, event response time is grater than zero, and time cannot move backwards. Causal ordering states that the order in which messages are delivered to the application is consistent with the *happens before* relation of the corresponding sending events. More formally:

Definition 2.2 *An execution of a distributed system σ respects causal order if for any two messages m_1 and m_2 sent by p_i and p_j such that $send_{p_i}(m_1) \rightarrow send_{p_j}(m_2)$, for every node q $del_q(m_1) \rightarrow del_q(m_2)$.*

A delivery mechanism implements causal ordering if every execution generated by it obeys Definition 2.2.

3 Causal Ordering Protocols

3.1 Timestamp Protocol

The protocol we describe here, nicknamed **Timestamp**, was originally introduced in [11] and later improved in [12]. Each node i maintains a vector of integers T_i with one entry per node. $T_i[j]$ indicates the sequence number of the last message i delivered from node j. Each message m_i is timestamped with vector T_i. When node i receives a message m with vector T, it compares vector T to its own vector T_i. If one or more messages that are causally prior to m have not arrived yet, then m is inserted into a *delivery_buffer* until the missing messages arrive. The pseudo code is given in Figure 1.

3.2 Symmetric Message Broadcast in an Overlay

This protocol, nicknamed **Overlay**, is simply constrained flooding across the logical arcs of the overlay topology. In other words, the idea is to broadcast each message along the overlay edges. Node i that wishes to send message m to all nodes in the system, sends the message only to its overlay neighbors. The first time node j receives message m, j sends m to all its overlay neighbors excluding the node from which it received m. Afterwards, j delivers m to the application.

The special way messages are being sent eliminates the need to send extra control information, while automatically gaining causal ordering between messages. This solution is regular, simple and saves complicated calculations time. Also, messages can always be delivered immediately to the application as soon as they are received from the network without any buffering. The pseudo code appears in Figure 2.

Theorem 3.1 *The protocol described in Figure 2 delivers messages in causal order.*

Notation
each node i maintains the following:
T_i – sequence number array of received messages
$delivery\_buffer_i$ – buffer of messages waiting to be delivered
$delivery\_buffer_i.insert(m)$ – insert message m to buffer
$delivery\_buffer_i.delete(m)$ – delete message m from buffer

Initialization of node i
T_i:=[0,0,0...]; $delivery\_queue_i$:= empty;

```
< At node i >
1:  upon receive(m) from node j do
2:      if (message_can_be_causally_delivered(m, j)) then
3:          handle_delivery(m);
4:          check_delivery_buffer()
5:      else
6:          delivery_buffer_i.insert(m)
7:      endif;
8:  done

9:  upon send(m) from application do
10:     T_i[i] := T_i[i] + 1;
11:     attach T_i to m;
12:     net_multicast(m)
13: done

14: message_can_be_causally_delivered(m, s)
15:     if ∀k ≠ s, m.T[k] ≤ T_i[k] and m.T[s] == T_i[s] + 1 then
16:         return true
17:     else
18:         return false
19:     endif

20: handle_delivery(m)
21:     deliver(m) to application;
22:     T_i[m.source] := m.sequence_number

23: check_delivery_buffer()
24:     scan delivery_buffer_i repeatedly
25:         for each mesage m in delivery_buffer_i do
26:             if (message_can_be_causally_delivered(m, m.source)) then
27:                 handle_delivery(m);
28:                 delivery_buffer_i.delete(m)
29:             endif
30:         done
31:     until no message is delivered to the application
```

Figure 1: **Timestamp** protocol – code for node p_i

Notation
each node i maintains the following:
$neighbors_i$ – node i's logical neighbors in the overlay
S_i – sequence number array of received messages

Initialization of node i
S_i:=[0,0,0...];

```
< At node i >
1:  upon receive(m) from node j do
2:      if (S_i[m.source] < m.sequence_number) then        // if we did not get m before
3:          deliver(m) to application;
4:          net_send(m) to every node in neighbors_i \ {j};
5:          S_i[m.source] := m.sequence_number
6:      else
7:          delete m
7:      endif
8:  done

9:  upon send(m) from application do
10:     S_i[i] := S_i[i] + 1;
11:     m.sequence_number := S_i[i];
10:     net_send(m) to neighbors_i
11: done
```

Figure 2: **Overlay** protocol – code for p_i

Proof: Assume, by way of contradiction, that there exists an execution σ of the protocol in which some messages are delivered in an order that violates causal ordering. Denote by q the first process that violates causal ordering in σ. Let m_1 and m_2 be the first two messages that q delivered in an order that violates causal ordering. Denote by p_i the process that sends m_1, and p_j be the process that sends m_2. Assume, w.l.o.g., that $send_{p_i}(m_1) \rightarrow send_{p_j}(m_2)$, but $del_q(m_2) \rightarrow del_q(m_1)$. Let q' be one of q's neighbors such that q received m_2 from q' for the first time. According to the protocol, delivery is done in the order messages are received for the first time. So the assumption $del_q(m_2) \rightarrow del_q(m_1)$ also implies $recv_q(m_2) \rightarrow recv_q(m_1)$. According to the assumption that q is the first process that delivered m_2 before m_1, q' received m_1 for the first time before it received m_2 for the first time. There are two possibilities.

If q' received m_1 for the first time from q then: q sent m_1 and thus delivered m_1 before it received and thus delivered m_2. More formally, $del_q(m_1) \rightarrow net\_send_q(m_1, q') \rightarrow recv_{q'}(m_1) \rightarrow recv_{q'}(m_2) \rightarrow net\_send_{q'}(m_2, q) \rightarrow recv_q(m_2) \rightarrow del_q(m_2)$. A contradiction to the assumption that $del_q(m_2) \rightarrow del_q(m_1)$.

If q' received m_1 for the first time from $q'' \neq q$ then: The assumption $recv_{q'}(m_1) \rightarrow recv_{q'}(m_2)$

Protocol	**Overlay**	**Timestamp**
Number of times message is being sent	$O(n \cdot \log(n))$	$O(n)$
Control information size	none	$O(n)$
Total bytes sent	$O(n \cdot \log(n) \cdot d)$	$O(n^2 + n \cdot d)$

Table 1: Analytical comparison of the causal ordering protocols

leads to $net\_send_{q'}(m_1, q) \rightarrow net\_send_{q'}(m_2, q)$. Because communication lines preserve FIFO, $recv_q(m_1) \rightarrow recv_q(m_2)$ and thus $del_q(m_1) \rightarrow del_q(m_2)$ A contradiction to the assumption that $del_q(m_2) \rightarrow del_q(m_1)$.

■

4 Analytical Comparison

A comparison of the two causal ordering protocols is presented in Table 1 assuming a hypercube overlay topology. In this comparison, we denote by d the size of the application data. In **Overlay**, each message is sent up to $n \cdot \log n$ times (n nodes receive it, and each send it to $\log n$ neighbors). No control information is added, so the total traffic generated by this protocol is $O(n \cdot \log n \cdot d)$. In the **Timestamp** protocol, each message is timestamped with additional control information of size n. The message is sent to n nodes, so the total protocol traffic is $O(n^2 + n \cdot d)$. We expect that when the size of application data is small, the **Overlay** protocol will be much better. This is because the overhead of the control information used by the **Timestamp** protocol will outweigh the redundant messages sent by the **Overlay** protocol. When the data is large, we expect the opposite to happen, since the vector will be insignificant in comparison to data size. Additionally, the redundant transmissions of **Overlay** become more expensive, since the transmission cost is proportional to the size of the message. More precisely, theoretically, we would expect **Overlay** to be better roughly when $d < n/(\log n - 1)$.

In terms of resiliency to process crash failures, the **Timestamp** protocol can be made to tolerate any number of failures. The number of failures that can be sustained by the **Overlay** protocol depends on the topology used. With a hypercube topology, it can overcome $\log n$ failures. If more failures occur, the logical arcs might need to be adjusted to bypass failed nodes. See additional discussion in Section 7.

As a final observation, **Overlay** has much simpler code. This property typically reduces programmers bugs.

5 Experimental Performance

5.1 Simulation Model

We used the ns [31] simulator to investigate the causal ordering protocols described in Sections 3.1 and 3.2. Our aim is to study the effect of the number of nodes on system performance. We measured the following indices for the two protocols:

Message delay: Message delay is the time between when a message is sent and its delivery. We measured both the mean message delay and maximum message delay. Each node keeps this information for each message it delivers.

Node delivery queue size: This is the size of queues of messages that are waiting to be delivered. This impacts the actual memory usage of each protocol, and accounts for part of the message delay. In the **Overlay** based protocol, a message can be delivered immediately after being received, and thus there are no such queues.

Network load: Here we measure the average network load, i.e., the average number of messages on all links in the network at any given time, and the maximum queue size, i.e., the maximum number of messages waiting to be sent on any of the links in the system.

Total number of messages: This measures the total number of messages sent. However, since our simulatur simulates a real IP network, we in fact measure the number of fragments sent. We also presents the total number of bytes sent. Each message (fragment) is counted once for each logical arc it traverses.

Hop count: This is the total number of physical hops that messages travel through. Each message is counted once at each physical hop that it passes on its way from source to destination. This index shows the overall protocol space requirements on all physical links in the system.

The causal ordering protocols were tested with several random generated network topologies created by the network generator GT-ITM [14] with edge probability varying from 0.01 to 0.04. Each protocol was run multiple times and in each run a different randomly generated topology was used. The results presented are the average on all runs. The links in the simulations were chosen to be duplex 10Mbps in each direction with uniformly distributed delay between 0 to 1 ms. We assumed that the number of senders is 10, and the total number of nodes varied between 100 to 600. The sequence number size was assumed to be 4 bytes, and thus the size of the timestamp vector carried by the **Timestamp** protocol message is $4 \cdot n$ bytes. Also, the message header size was set to 32 bytes, large enough for most transport protocols [3, 39]. In our experiment, each sender node initiates 10 new messages every 1 milliseconds. We measured the results for three message sizes: 0 bytes, 100 bytes, and 1000 bytes.

In the **Overlay** protocol, the logical hypercube was built according to node id's binary representation. Since the network was generated randomly, the fitness of logical structure to

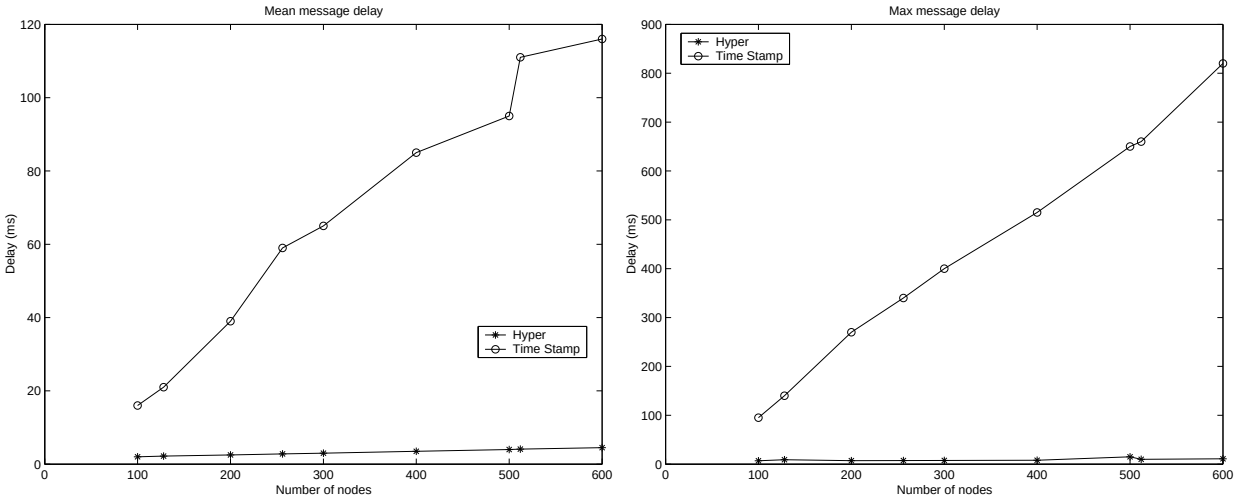

Figure 3: Message delay as a function of system size when data size is 0

the network is random as well. Also, in **Overlay**, when the number of nodes is not a power of two, we used the construction of incomplete hypercubes described in [19, 30].

5.2 Simulation Results

Message delay The mean and maximum message delay from source to destination is reported in Figures 3, 4, and 5. When the data size is 0, the **Overlay** protocol delivers messages much faster. When data size is 100, the difference between **Overlay** and **Timestamp** decreases. Finally, when data size is 1000, **Timestamp** becomes faster than **Overlay**. Notice that the message delay of **Overlay** is more influenced by message data size than **Timestamp**. As we discussed before, this is because **Overlay** sends each message $\log n$ times more than **Timestamp**.

This is echoed when looking at the mean queues at links (Figures 15, 16 and 17) and the mean delivery queues of nodes (Figures 12, 13, 14). In the **Timestamp** protocol, messages are delayed before delivery so message delay is affected both from the network delay, which largely depends on link queues, and from the delay within the node itself. In the **Overlay** protocol, messages are not delayed in the nodes before being delivered and thus only link queues affect their latency.

Total number of messages The total number of messages sent and total bytes sent are reported in Figures 6, 7 and 8, while the hop count results are reported in Figures 9, 10 and 11. When looking at the message count and hop count graphs, it appears that the **Timestamp** protocol generates fewer messages. This is because in the **Timestamp** protocol each message

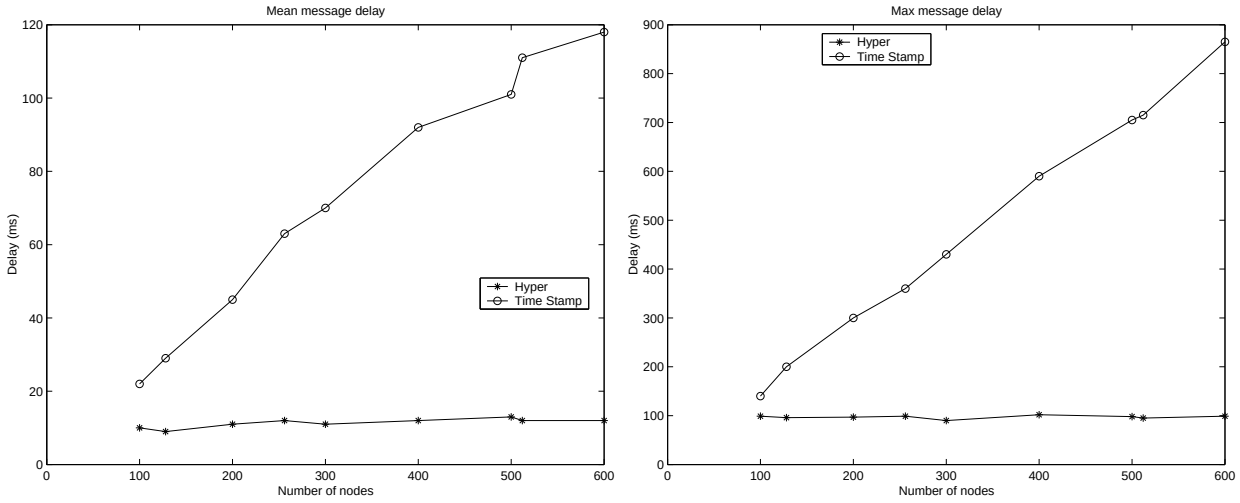

Figure 4: Message delay as a function of system size when data size is 100

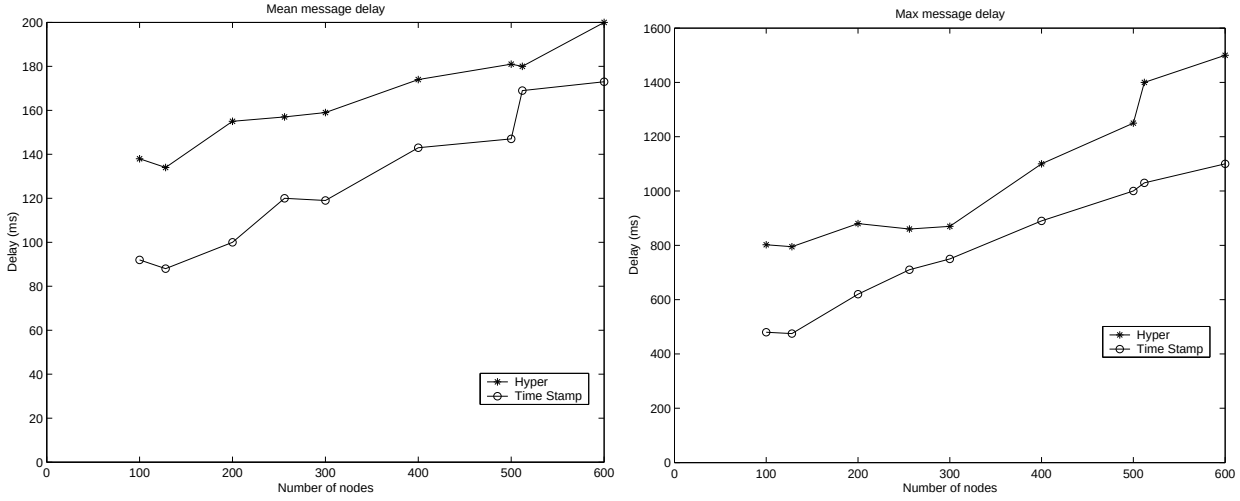

Figure 5: Message delay as a function of system size when data size is 1000

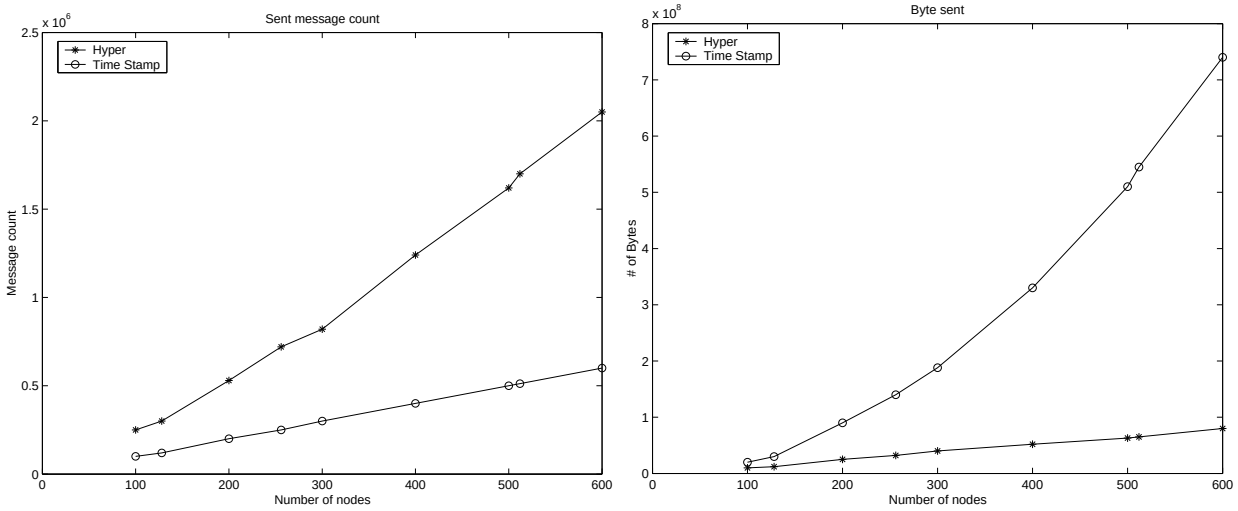

Figure 6: Message count as a function of system size when data size is 0

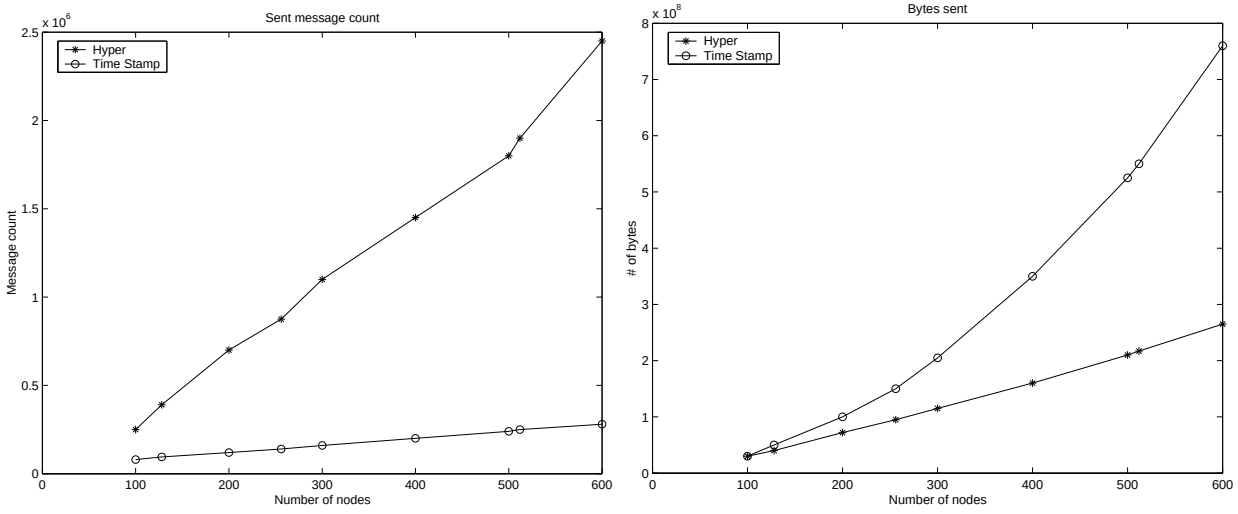

Figure 7: Message count as a function of system size when data size is 100

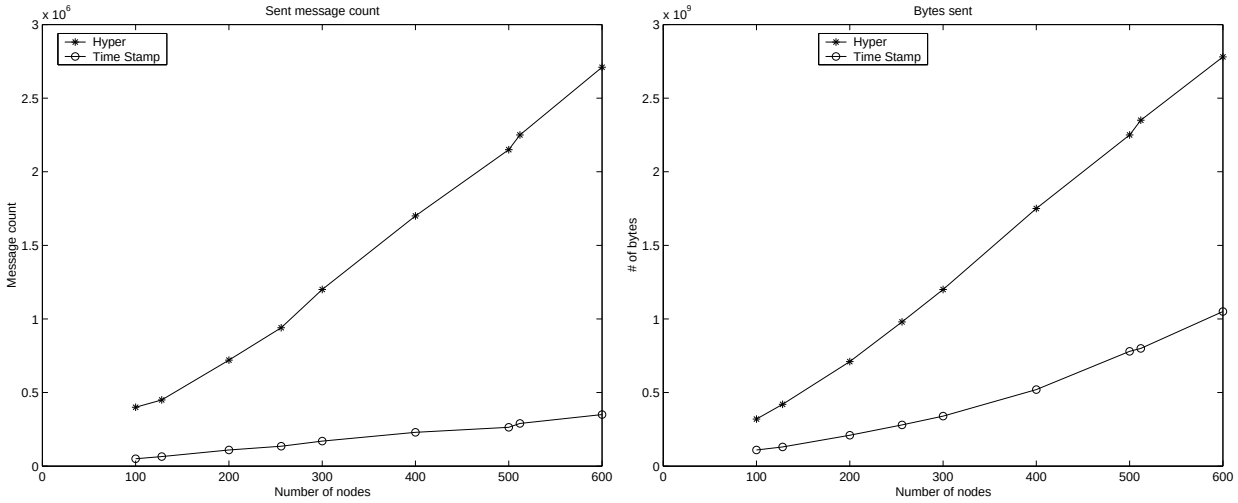

Figure 8: Message count as a function of system size when data size is 1000

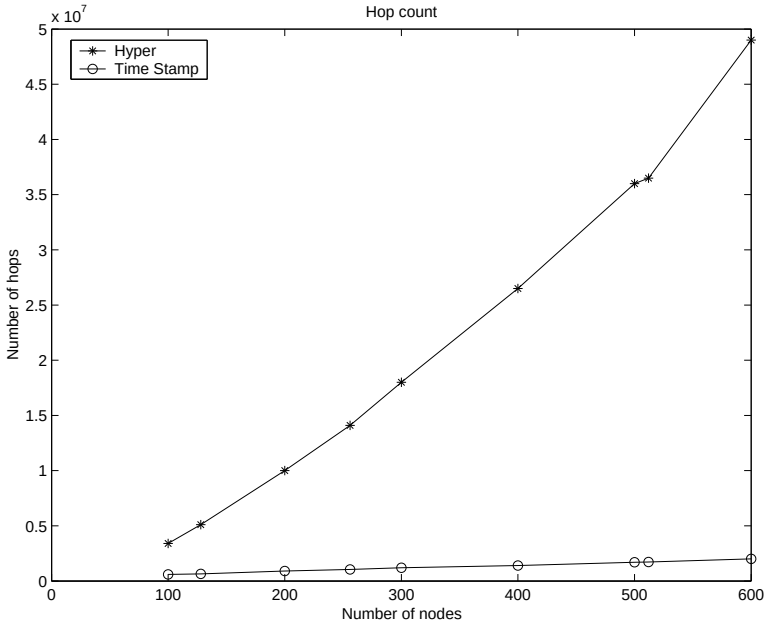

Figure 9: Hop count as a function of system size when data size is 0.

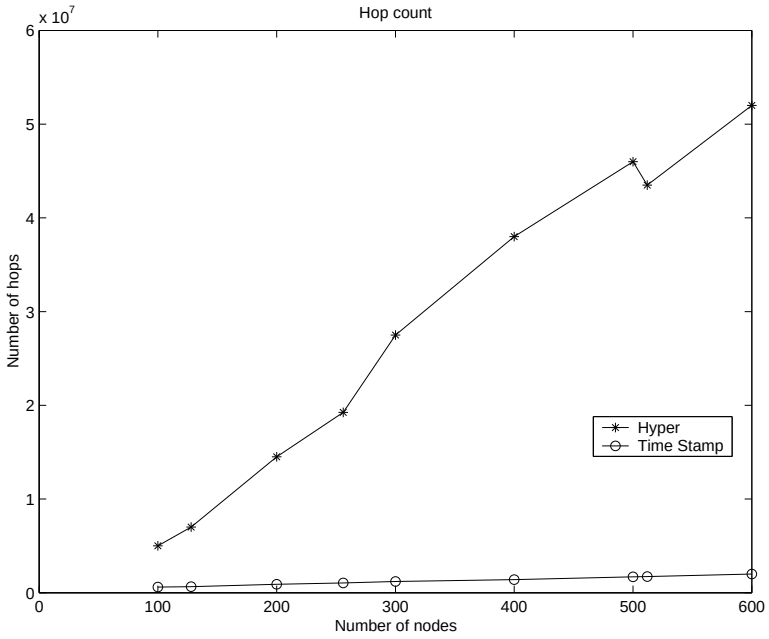

Figure 10: Hop count as a function of system size when data size is 100.

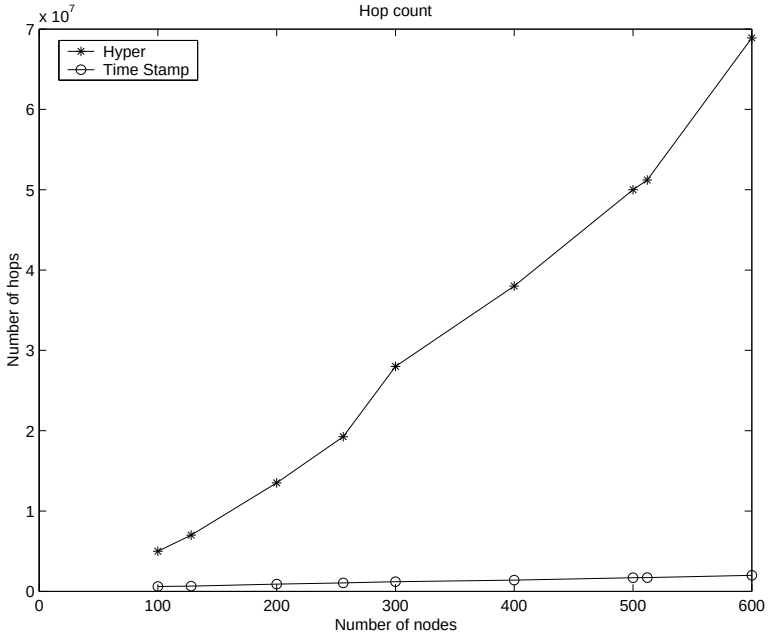

Figure 11: Hop count as a function of system size when data size is 1000.

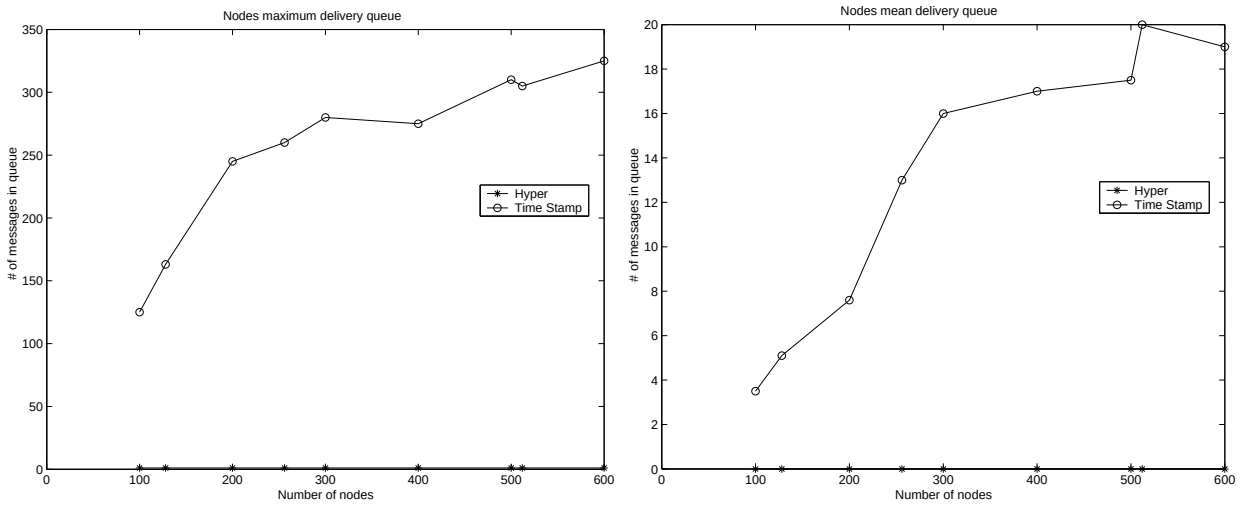

Figure 12: Node message queue size a function of system size when data size is 0

is sent once to all nodes, and thus counted as n sent messages; In the **Overlay** protocol, each message is sent $n \cdot logn$ times.

Yet, the real overhead of the protocol is reflected in the bytes sent graphs. As described in section 4, the **Overlay** protocol uses less control information when the message data size $d < n/(\log n - 1)$. This theoretical diagnostic is concluded from the **Overlay** protocol worst case in which each message is sent $n \cdot \log n$ times and thus uses $n \cdot \log n \cdot d$ bytes. However, the average case is better and thus **Overlay** uses fewer bytes even when data size is 100. When data size is 1000, the **Overlay** protocol sends more bytes than **Timestamp** as predicted by the formula described above.

Node delivery queue size The maximum and average size of delivery queues is reported in Figures 12, 13, and 14. The **Overlay** protocol does not delay messages before delivery and thus the message delivery queues are always empty. On the other hand, the **Timestamp** protocol delays messages in the queue until they can be causally delivered. It can be seen that data size has little affect on the maximum queue size and mean queue size; this is because when messages are larger, all messages arrive after proportionally longer time. In particular, we measure the number of message in the delivery queues and not the time they spend in the network. (The latter is reflected in the measurements of message delay.) On the other hand, when the number of nodes increases, the delivery queues become longer, which hurts the scalability of the **Timestamp** protocol.

Network load The average measured router queue size and the maximum queue size are reported in Figures 15, 16 and 17. When data size is 0, the **Timestamp** protocol has larger mean queue size because it uses more bytes (Figure 6). In the **Overlay** protocol, the links

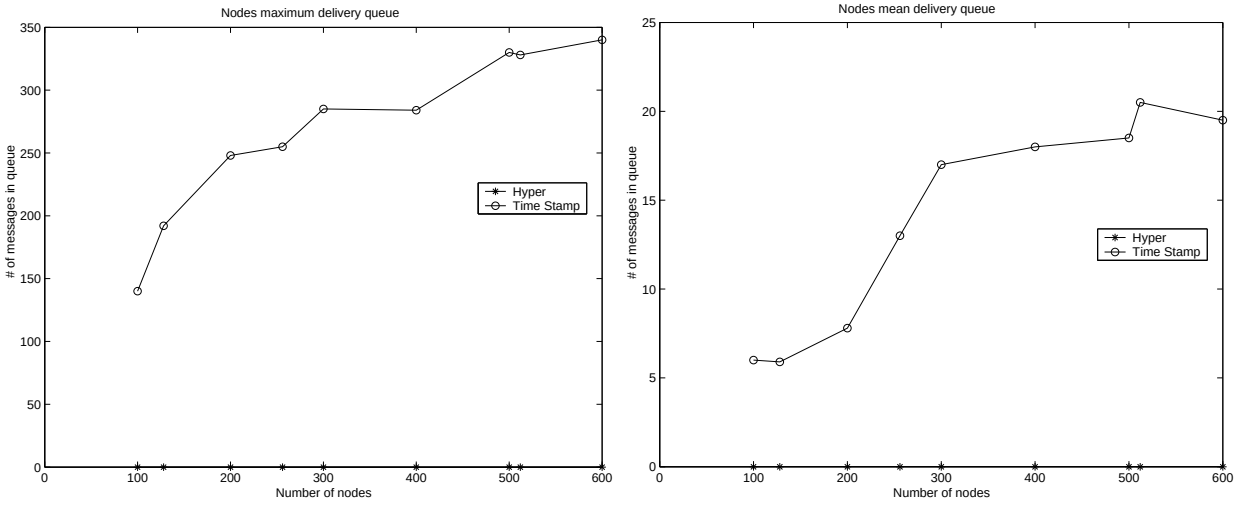

Figure 13: Node message queue size a function of system size when data size is 100

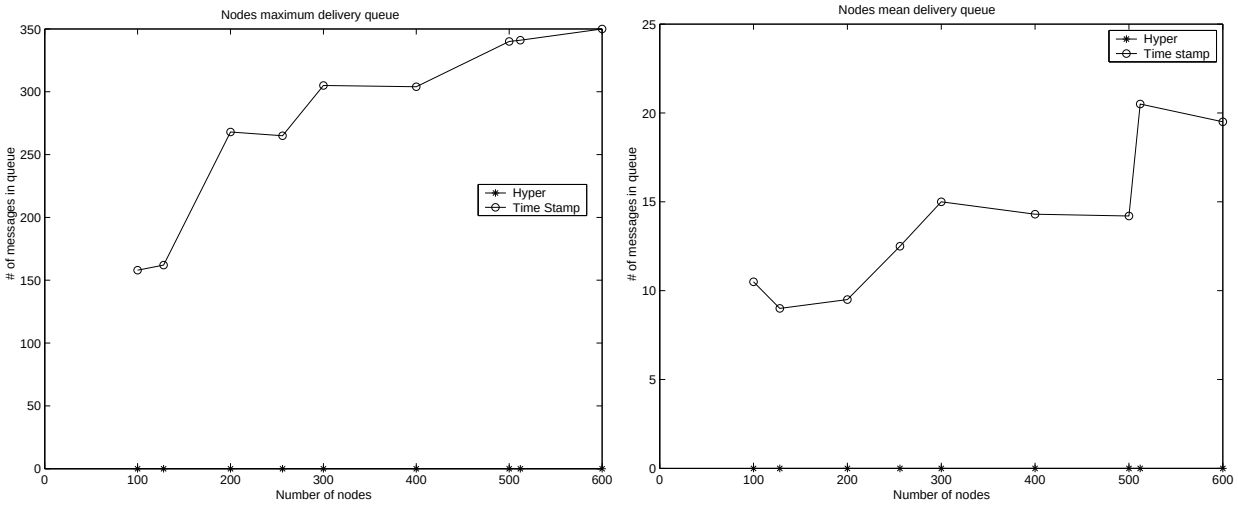

Figure 14: Node message queue size a function of system size when data size is 1000

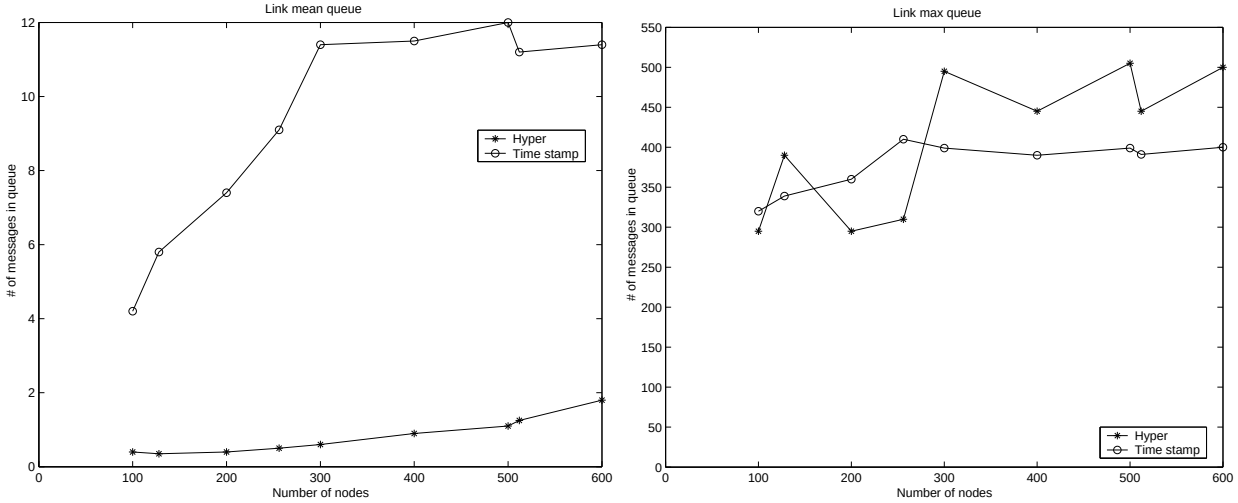

Figure 15: Network load as a function of system size when data size is 0

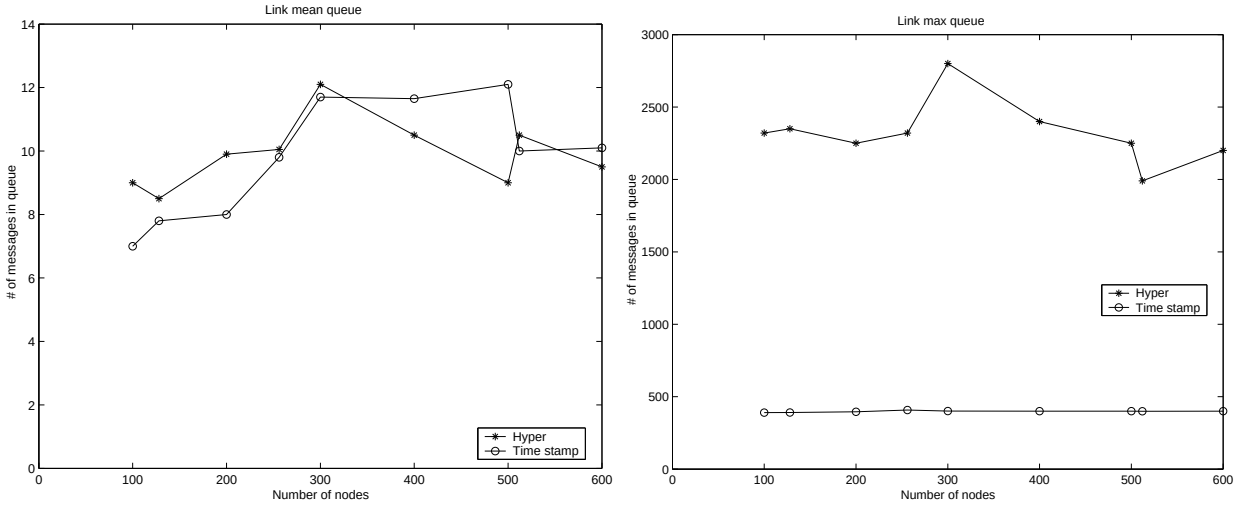

Figure 16: Network load as a function of system size when data size is 100

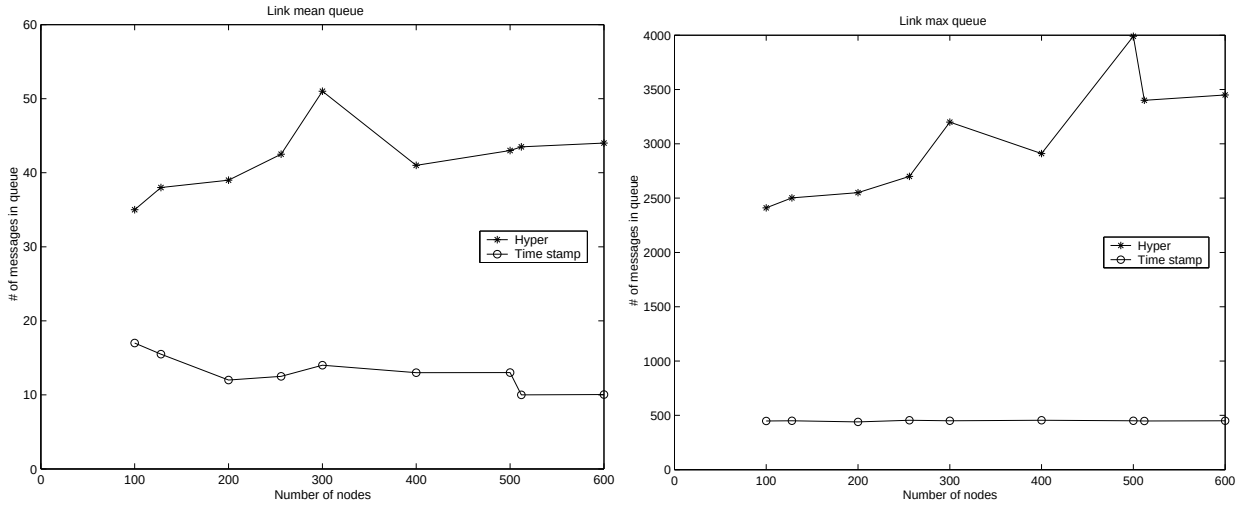

Figure 17: Network load as a function of system size when data size is 1000

that are connected to the senders are loaded for short periods. This is because the senders use point to point messages that load their links when they send messages to all their neighbors at once. After the messages pass the first links, they spread in the network. The result is high maximum queue size and low mean queue size. When data size increases, the **Overlay** protocol mean and maximum queues increase in comparison to the **Timestamp** protocol. It reflects the increase in the amount of data sent (Figures 7 and 8).

6 Related Work

The first algorithm for causal ordering, initially implemented on ISIS, piggybacks unstable messages on each new message. While this algorithm does not include control information, it incurs the overhead of resending each message. When optimized, this protocol reduces to constrained flooding over a flat clique architecture. In a flat architecture, each message is sent $O(n^2)$ times, which is why it is less scalable than using the hypercube topology.

The timestamp algorithm we described here appeared first in [11]. That algorithm was later extended to support point-to-point messages (and multicasts to proper subsets of the nodes) in [36] by using matrix timestamps. The problem with this extension is that it piggybacks $O(n^2)$ control information on each message. More recent works, like [5, 25, 33], reduce the typical control overhead by sending only the ids of the last unstable message from each process that the new message depends on. However, in the worst case, the control information overhead is still $O(n^2)$.

Another approach for scalable causal ordering is to construct hierarchies, as proposed in [2, 8, 35]. The work of [35] is based on dividing the problem into physical domains, where in

each domain the gateway acts as a *causal separator* and is responsible for forwarding messages correctly to other domains. On the other hand, the hierarchy in [2, 8] need not match a specific physical topology, and group communication technology is used to maintain *local groups* and reliably elect their representative in the higher levels of the hierarchy.

Hypercubes were originally proposed as an efficient interconnect for massively parallel processors (MPP) [23, 26]. A great body of research has been done in solving parallel problems on hypercubes, as described in [23, 26]. In particular, much work has been done in the area of routing, one-to-all and all-to-all communication and gossiping in hypercubes [9, 15, 16, 17, 24, 37].

The idea of using logical hypercube topologies for reliable multicast was explored in our work on stability detection [19], and in the HyperCast toolkit [27]. The HyperCast work does not address causal ordering. Also, with the HyperCast protocol, some nodes in an incomplete hypercube have smaller degree than others, which reduces its fault-tolerance. For example, when the hypercube has $2^n + 1$ nodes, one of the nodes has only a single neighbor according to HyperCast. In contrast, our construction guarantees that even for incomplete hypercubes, the degree of each node and the number of node independent paths between each pair of nodes is roughly $\log(n)$ [19, 30].

Recently, there have been several bodies of work on generating an approximation of hypercube topologies in a fully distributed manner, and providing efficient routing and lookup services in these overlay networks [29, 32, 38, 42]. These mechanisms can be used, for example, as an infrastructure for large scale publish/subscribe systems. They are very scalable and adaptable, but providing causal ordering more efficiently than total ordering in these frameworks is still an open issue.

7 Conclusions

In this paper, we have formally proved that constrained flooding along the arcs of a deterministic logical topology preserves causal ordering at no additional cost. We have also analyzed the performance of this approach when implemented over a hypercube topology, and compared it to the standard vector timestamp based implementation of causal ordering. The results indicate that such a scheme is much more scalable for small messages, as is typical in many control applications, notification systems, and stock quotes. However, as expected, when the messages become large, the timestamp protocol performs better, since the size of messages becomes more significant than the amount of control information, which grows linearly with the number of nodes in that protocol.

An important issue that we have not addressed in this paper is how to handle dynamic changes in the system. As a proof of concept, we have recently implemented a prototype of this scheme in the Ensemble group communication toolkit [22]. That implementation enjoys the benefits of a group membership service, and relies on four simplifying assumptions that Ensemble satisfies due to its strong virtual synchrony model [20], but at a cost. First, when a network router dies, either the network recovers in a timely manner, or Ensemble will partition

the group, after which causal ordering need only be maintained within each partition. Second, when a node dies, Ensemble eventually detects the failure and installs a new view from which that node is excluded. (As discussed in Section 4, the protocol runs well without configuration changes as long as fewer than $\log n$ nodes fail. This allows setting a long failure detection timeout for Ensemble.) Third, at any given time, all nodes agree on the membership, which has the benefit that each node can compute its neighbors accurately. Fourth, messages do not cross view boundaries, which imposes the restriction that the application cannot send new messages during view changes [20]. The latter assumption is necessary for our correctness proof to hold. Clearly, the third and fourth guarantees above are costly to implement, and might hinder the scalability of the system beyond a few hundred nodes, or even dozens of nodes over a WAN. Therefore, generalizing the algorithm to overcome dynamic changes directly is an important open problem.

References

[1] The JavaGroups Project. http://www.javagroups.com.

[2] N. Adly and M. Nagi. Maintaining Causal Order in Large Scale Distributed Systems Using a Logical Hierarchy. In *Proc. of the 12th IASTED International Conference on Applied Informatics*, pages 214–219, February 1995.

[3] R. Ahuja, S. Keshav, and H Saran. Design, Implementation, and Performance of a Native Mode ATM Transport Layer. *IEEE/ACM Transactions on Networking*, 4(4):502–515, August 1996.

[4] S. Alagar and S. Venkatesan. An Optimal Algorithm for Distributed Snapshots with Causal Message Ordering. *Information Processing Letters*, 50:311–316, 1994.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[6] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[7] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-2, The Center for Networking and Distributed Systems, Computer Science Department, John Hopkins University, 1998.

[8] R. Baldoni, R. Beraldi, R. Friedman, and R. van Renesse. The Hierarchical Daisy Architecture for Causal Delivery. *Distributed Systems Engineering Journal*, 6:71–81, 1999.

[9] O. Baudon, G. Fertin, and I. Havel. Routing Permutations and 2-1 Routing Requests in the Hypercube. *Discrete Applied Mathematics*, 113(1):43–58, 2000.

[10] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, , and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[11] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[12] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[13] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.

[14] K. Calvert and E. Zegura. GT-ITM Random Network Generator. http://www.cc.gatech.edu/projects/gtitm.

[15] B.S. Chlebus, K. Diks, and A. Pelc. Optimal Broadcasting in Faulty Hypercubes. In *Proceedings of 21st Annual International Symposium on Fault-Tolerant Computing*, pages 266–273, Montreal, Canada, June 1991.

[16] B.S. Chlebus, K. Diks, and A. Pelc. Fast Gossiping with Short Unreliable Messages. *Discrete Applied Mathematics*, 53:15–24, 1994.

[17] R. Feldmann, J. Hromkovic, S. Madhavapeddy, B. Monien, and P. Mysliwietz. Optimal Algorithms for Dissemination of Information in Generalized Communication Networks. In *Proceedings of Parallel Architectures and Languages Europe*, pages 115–130. Springer, 1992. Lecture Notes in Computer Science, 605.

[18] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[19] R. Friedman, S. Manor, and K. Guo. Scalable Hypercube Based Stability Detection. *IEEE Transactions on Parallel and Distributed Systems*, 13(8), August 2002.

[20] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 140–149, October 1996.

[21] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Department of Computer Science, Cornell University, 1998.

[22] M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.

[23] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[24] D.W. Krumme. Fast Gossiping for the Hypercube. *SIAM Journal on Computing*, 21(2):365–380, April 1992.

[25] A. D. Kshemkalyani and M. Singhal. Necessary and Sufficient Conditions on Information for Causal Message Ordering and Their Optimal Implementation. Technical Report 29.2040, IBM Corporation, July 1995.

[26] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Kaufmann, 1992.

[27] J. Liebeherr and T.K. Beam. HyperCast: A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology. In *Proceedings of 1st International Workshop on Networked Group Communication (NGC '99)*, pages 72–89. Springer, July 1999. Lecture Notes in Computer Science, 1736.

[28] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus Deterministically Constrained Flooding on Small Networks. In *Proc14th International Conference on Distributed Computing 2000*, pages 253–267, October 2000.

[29] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing*, August 2002.

[30] S. Manor. Scalable Multicast in a Logical Hypercube. Master's thesis, Department of Computer Science, Technion – Israel Institute of Technology, August 1999.

[31] S. McCanne and S. Floyd. NS (Network Simulator) Home Page. http://www-nrg.ee.lbl.gov/ns.

[32] C. Plaxton, R. Rajaram, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[33] R. Prakash, M. Raynal, and M. Singhal. An Efficient Causal Ordering Algorithm for Mobile Computing Environment. *Journal of Parallel and Distributed Computing*, 41(2), March 1997.

[34] M. Raynal, A. Schiper, and S. Toueg. The Causal Ordering Abstraction and a Simple Way to Implement It. *Information Processing Letters*, 39:343–350, 1991.

[35] L. Rodrigues and P. Verissimo. Causal Separators and Topological Timestamping: an Approach to Support Causal Multicast in Large-Scale Systems. In *Proc. of the 15th International Conference on Distributed Computing Systems*, May 1995.

[36] R. Schwarz and F. Mattern. Detecting Causal Relations in Distributed Computing: in Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[37] D.S. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of 6th Distributed Memory Concurrent Computers Conference*, pages 398–403, 1991.

[38] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, August 2001.

[39] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proc. ACM SIGCOMM'96*, pages 96–104, August 1996.

[40] R. van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[41] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip Style Failure Detection Service. In *IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 55–70, April 1998.

[42] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley, April 2001.