On the Cost of Prioritized Atomic Multicast Protocols

Emili Miedes, Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia (SPAIN)

{emiedes,fmunyoz}@iti.upv.es

Technical Report ITI-SIDI-2009/002

On the Cost of Prioritized Atomic Multicast Protocols

Emili Miedes, Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia (SPAIN)

Technical Report ITI-SIDI-2009/002

e-mail: {emiedes,fmunyoz}@iti.upv.es

February 2009

Abstract

A prioritized atomic multicast protocol allows an application to tag messages with a priority that expresses their urgency and tries to deliver first the ones with a higher priority. For instance, such a service can be used in a database replication context, to reduce the transaction abort rate when integrity constraints are used. We present a study of the three most important and well-known classes of atomic multicast protocols in which we evaluate the cost imposed by the prioritization mechanisms, in terms of additional latency overhead, computational cost and memory use. This study reveals that the behavior of the protocols depends on the particular properties of the setting (number of nodes, message sending rates, etc.) and that the extra work done by a prioritized protocol does not introduce any additional latency overhead in almost all of the settings evaluated. This study is also a performance comparison of these classes of total order protocols and can be used by system designers to choose the proper prioritized protocol for a given setting.

KEYWORDS: Broadcast protocols, atomic broadcast, total order broadcast, priority management

NOTE: This report supersedes the previous TR-ITI-ITE-08/17 report.

1 Introduction

A group communication service (GCS) is a middleware component that provides a set of services that can be used as building blocks to design and build distributed systems. A GCS usually offers an atomic (i.e., total order) multicast message delivery service which enables an application to send messages to a set of destinations such that they are delivered in the same order to each destination. Group communication and total order topics have been studied for more than two decades from both a theoretical [4, 6] and a practical [3, 7, 12, 1] point of view. A useful additional guarantee a GCS may offer is priority-based delivery [16, 15, 13], which allows a user application to prioritize the sending and delivery of certain messages.

Such a service can be used in a scenario like the following. Consider an application that runs on top of a database replication system and is physically distributed among several sites. Such systems usually follow a *constant interaction* model [18], according to which, updates made by a transaction are broadcast in total order to all the database replicas at the end of the transaction, using a single message. The order in which a set of messages corresponding to different transactions are delivered by the replicas determines the final order in which a set of transactions are applied to the database. This order has a deep impact on the evaluation of the integrity constraints defined in the database. The idea is to alter the order in which transactions are committed for achieving a favorable constraint evaluation, thus reducing the transaction abort rate. Note that the database replication protocol is able to know which database tables and fields have been accessed by a given transaction, and it is able to use such information for assigning priorities. To do so, the replication protocol should be also aware of the semantic integrity constraints defined in the database schema. MADIS [8] is an example of database replication middleware where all these issues can be managed. A transaction implementation based on stored procedures is another alternative for providing all the information needed by the replication protocol in order to assign priorities (accessed tables and fields, values being used in the updates, etc.).

Non-prioritizing total order broadcast policies have been widely studied, while, as far as we know, only a few studies exist for priority-based protocol variants. In [15] and [13], two priority-based total order protocols are presented. Low priority messages may suffer starvation if too many high priority messages are sent. The problem of message starvation is dealt with specifically in [14]. In [2, 17] another common problem of this kind of protocols, known as *priority inversion*, is addressed.

We studied recently atomic multicast prioritization from both a theoretical and a practical point of view. In [11] we presented some techniques to modify existing total order broadcast protocols to take into account message priorities. We also showed how these techniques can be applied to existing total order protocols and identified which technique is the most suitable for each of the classes of total order broadcast protocols presented in [6]. Then, in [10] we proved that total order prioritization is able to reduce transaction abort rates in replicated databases, thus showing the utility of atomic multicast prioritization. In this paper we show that atomic multicast prioritization techniques do not impose a significant overhead on the latency of the multicast messages. As a result, this reinforces the usefulness of this approach, since its advantages proved in [10] do not introduce any performance degradation.

The paper is organized as follows. In Section 2 we describe the system we use in this experimental work. In Section 3 we present some experimental work we have done to show that the overhead added by the prioritization techniques is not significant. Finally, we conclude the paper in Section 4. Sections 1 to 4 are included in a paper that we are going to send to a conference. In Appendix A we include additional results that could not be included in that paper, due to space limitations.

2 System Description

The system is composed of a set of processes that communicate through message passing. Each process has a multilayer structure, whose topmost level is a user application that accesses a replicated DBMS, which in turn uses the services offered by a group communication system. The latter is composed of one or more group communication protocols, which use the underlying network's services to send and deliver messages. In Section 3.1 we provide additional information related to the physical environment we used.

Processes run on different physical nodes and the drift between two different processors is not known. The time needed to transmit a message from one node to another is bounded but the bound is unknown. In practice, the system does not need more synchrony than that offered by a conventional network which offers a reasonably bounded message delivery time. Process failures and network partitions may occur. However, since we are focusing on the comparison of prioritization techniques, we do not address failure handling (which can be realized by mechanisms such as group membership services and fault-tolerance protocols).

3 Experimental Work

In this section, we present the experimental work we have done to observe the performance of the total order protocols and evaluate the cost overhead of their prioritized versions. First of all, we describe the testbed, including the physical setting. Then, we describe the parameters and the methodology used to run the tests and finally we present and discuss the results.

3.1 Testbed

To evaluate the prioritization techniques, we implemented three total order protocols: a sequencer-based, a privilege-based and a communication history one. We also implemented their corresponding prioritized versions, according to the techniques proposed in [11]. To analyze the performance of the total order protocols, we use a test application that uses the services of a total order protocol which in turn uses a reliable transport layer.

The experiments have been conducted in a system of eight nodes with an Intel Pentium D 925 processor at 3.0 GHz and 2 GB of RAM, running Debian GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 24-port 100/1000 Mbps DLINK DGS-1224T switch that keeps the nodes isolated from any other node, so no other network traffic can influence the results.

3.2 Methodology

To evaluate the performance of the prioritization techniques, in each node, the application broadcasts a series of messages to all the nodes in the system, by means of a total order protocol. The messages are broadcast at a uniform sending rate which is constant during the whole test. As explained below, we have performed tests with different sending rates. Besides this, we have no other flow control mechanism neither in the application nor in the total order protocols.

Each message is tagged with a uniformly-distributed random priority which is an integer number. As discussed later, prioritized total order protocols use this value to prioritize the delivery of some messages over others.

The length of the messages is not fixed, but depends on the headers saved in them by the total order protocols. Nevertheless, in all the cases it is less than the MTU of the network we are using (1500 bytes), so all the application messages fit into one wire-level packet.

Each message is totally ordered and delivered by all the nodes in the system. To evaluate the performance of a given protocol, we measure the *delivery time* of each message, i.e., the time observed by the application in a given node, between the moment in which it broadcasts the message and the moment in which it receives back the message, once totally ordered.

For each message we have a delivery time and for each node we have a series of delivery times, corresponding to all the messages sent by that node. If we merge all the delivery times from all the nodes, we can compute a global mean and median delivery time. Such a mean (median) time expresses the mean (median) time needed by messages to get totally ordered.

This test is run with different total order protocols and also with their corresponding prioritized versions. With these values we analyze the dispersion of the series of delivery times. A significant difference between the mean and the median values, especially when the median is lower than the mean, implies that there is a number of (low priority) messages that have a high delivery time, which means that the prioritization mechanism is working as expected and has been able to prioritize a number of messages. Nevertheless, the mean value of the test should not exceed some bound. An excessively high value for the mean delivery time implies that too many messages are being delayed and this delay is extending their delivery times. In this case we say that the protocol became *saturated*.

In order to get more trustworthy results, we discard the first 3200 messages[1] recorded in each node. These values correspond to delivery times of messages delivered during a period of time in which the total order protocol is being initialized so the system is not yet in a steady-state regime.

During the execution of these tests we also analyzed two additional indicators: a) the processing time employed by the prioritization mechanisms and b) the memory use. In Section 3.4 we provide additional details.

3.3 Parameters

The considered parameters are the class of total order protocol, the number of nodes and the sending rate at which the test application broadcasts messages. These are described in the following subsections. We also include a discussion about the number of messages to be delivered by each node.

[1]This number has been chosen empirically, after analyzing the behavior of the data structures managed by the total order protocol implementations.

Protocol type. We have implemented three non-prioritized total order protocols and a prioritized version for each. The UB protocol is an implementation of the UB sequencer-based total order algorithm proposed by [9][2]. The TR protocol implements a token ring-based algorithm. It is similar to the ones of [12] and [1] and the TR protocol implemented in [11] but there is a significant difference. In the TR protocol, when a node receives the token, it broadcasts just a message, as in [5], instead of broadcasting a number of messages, as in [12] and [1]. Finally, the CH protocol is an implementation of the causal history algorithm in [6].

The corresponding prioritized versions are UB_{prio}, TR_{prio} and CH_{prio}, respectively. They have been implemented according to the *priority sequencing*, *priority sending* and *priority delivering* techniques proposed in [11], respectively.

Number of nodes. The application has been run with 4 and 8 nodes, each node running in a different physical node of the cluster.

We decided not to evaluate bigger systems for several reasons. First of all, we must consider our base context. The system is a replicated database environment and our final goal is to have proper tools that ease data replication and offer a high degree of data availability. In such a context, the use of more than three or four data node replicas is questionable. On the other hand, scalability is not a concern of this paper. We are only interested in the overhead being introduced by the prioritization techniques and such overhead does mainly depend on the behavior of the total order protocol taken as its basis.

Sending rate. In each test, a node broadcasts messages using a uniform sending rate. We have run tests with 4 and 8 nodes and sending rates of 10, 40, 60, 80 and 100 messages sent per second and node. Note that this generates maximum global sending rates of 400 msg/s and 800 msg/s, in systems with 4 and 8 nodes, respectively.

Number of messages delivered by each node. To ease the comparison, in each test, each node receives the same sequence of messages. This sequence has 32000 messages. A test ends when all the nodes deliver those messages.

To ensure a stable operation of the protocols during a test, each node sends more messages than those strictly necessary. For instance, in a test with 4 nodes, each node would only need to send 8000 messages. In practice, as the nodes deliver messages at a rate lower than the sending rate, there is a final period of time in a test in which the system is no longer *stable*, because the queues of the protocols are getting empty and this may affect the measuring of the delivery times. Moreover, the difference between the sending rate and the delivery rate is different in each test, and depends basically on all the parameters (the protocol used in the test, the number of nodes and the sending rate itself) and this poses additional difficulties to the protocol comparison.

To solve this issue, each node sends as many messages as needed, to ensure a continuous flow of messages during the whole test. This approach also solves the lack of liveness shown by the CH and CH_{prio} protocols, as described in [6].

3.4 Cost Evaluation

To evaluate the cost employed by the prioritization mechanisms, for each original protocol and its corresponding prioritized version we measure the time employed to run certain parts of both protocols. We call this time the *prioritization time*. The sections measured are semantically equivalent, so we can get comparable measures.

For instance, to evaluate the sequencer-based protocols, we measure the time lapse between the time when the sequencer starts to handle a message and the time when it broadcasts the message, once sequenced. The corresponding prioritized protocol has an equivalent section, in which prioritization takes place. Measuring the time needed to run both sections and comparing both times, we can get a very tight approximation of the time needed by the prioritization mechanism applied by the prioritized protocol.

[2]UB stands for *Unicast-Broadcast*, as in [6].

		UB	UB_{prio}	TR	TR_{prio}	CH	CH_{prio}
		4 nodes					
	mean	1.45	1.25	6.69	6.33	76.77	77.00
10	1st q.	1.20	1.08	0.89	0.89	65.13	65.16
msg/s	med.	1.28	1.18	1.26	1.28	81.10	81.35
	3rd q.	1.36	1.26	9.30	7.52	93.38	93.44
	mean	1.50	1.46	1.29	1.27	17.77	17.86
40	1st q.	1.11	1.09	0.72	0.72	13.13	13.10
msg/s	med.	1.24	1.31	1.02	1.02	17.12	17.01
	3rd q.	1.34	1.54	1.27	1.27	20.84	20.84
	mean	1.30	1.51	1.70	1.70	12.22	11.95
60	1st q.	0.97	1.09	0.75	0.76	8.83	8.77
msg/s	med.	1.09	1.32	1.07	1.08	12.60	12.61
	3rd q.	1.24	1.53	1.32	1.35	12.88	12.91
	mean	3.43	2.20	2.36	2.75	9.13	9.29
80	1st q.	1.17	1.27	0.87	0.77	4.97	4.96
msg/s	med.	1.27	1.42	1.20	1.09	8.66	8.62
	3rd q.	1.53	1.70	1.51	1.37	8.98	8.96
	mean	134.25	487.36	4.85	26.14	7.10	6.71
100	1st q.	1.12	1.35	0.83	0.83	4.62	4.59
msg/s	med.	1.28	1.6	1.17	1.18	4.84	4.83
	3rd q.	1.79	2.75	1.51	1.52	5.14	5.20
		8 nodes					
	mean	1.89	11.35	2.05	2.08	90.33	90.96
10	1st q.	1.34	1.53	1.37	1.39	85.21	85.40
msg/s	med.	1.53	1.73	1.86	1.87	97.62	94.21
	3rd q.	1.70	1.92	2.39	2.4	101.79	101.74
	mean	3.84	221.37	7.85	7.60	23.62	23.20
40	1st q.	1.40	1.65	1.53	1.50	20.76	20.74
msg/s	med.	1.62	1.93	2.19	2.17	21.18	21.17
	3rd q.	2.04	2.86	2.91	2.86	21.89	21.68
	mean	190.82	670.48	75.53	151.49	17.09	17.22
60	1st q.	1.42	1.72	1.68	1.69	12.96	12.98
msg/s	med.	1.86	2.51	2.54	2.53	13.28	13.27
	3rd q.	3.65	9.94	3.69	3.60	17.07	17.05
	mean	6718.52	13608.62	460.35	750.16	86.96	136.80
80	1st q.	6373.32	13.32	2.24	2.21	9.02	9.18
msg/s	med.	6660.33	604.56	3.8	3.70	9.88	13.80
	3rd q.	6882.63	24776.30	340.26	34.26	65.51	237.24
	mean	20102.49	25264.85	5477.03	5148.22	100.05	125.82
100	1st q.	14290.93	104.60	5119.25	5.14	5.78	5.70
msg/s	med.	18435.50	17349.36	5517.79	65.87	9.27	9.64
	3rd q.	23159.88	47670.92	5891.15	6908.98	58.16	145.75

Table 1: Delivery times (ms) with 4 and 8 nodes

These measures are only comparable between a given protocol and its corresponding prioritized version. For other protocol families, the parts of the protocols considered are different.

For each test, we measure the prioritization time in each node[3]. Then we compute the mean prioritization time as the mean for all the nodes. These numbers are presented in great detail in Appendix A and summarized in Section 3.5.

To evaluate the memory use, we analyzed how much of the total amount of memory available by the Java Virtual Machine is being used during each test by each node. In Appendix A we graphically represent this evolution in several settings (in systems of different sizes, with different protocols and sending rates, as explained in 3.3). Moreover, for each test, we count the number of times the Java garbage collector has been run in each node and with all of them, we compute the mean number of garbage collection runs. These numbers are summarized in Section 3.5.

3.5 Results

For each test we computed a global mean and median delivery time, as explained above, and the corresponding first and third quartile, as well. The results are summarized in Table 1 and discussed in Section 3.6.

In Tables 2 and 3 we show the mean prioritization time and the mean number of garbage collection runs, computed as explained in Section 3.4.

3.6 Discussion

In Table 1 we show the mean and median global delivery times (in ms), as well as the first and third quartiles in systems with 4 and 8 nodes, respectively, at different sending rates.

[3]We also discard the first 3200 messages, as explained in Section 3.2.

In a system with 4 nodes, the UB and UB_{prio} protocols perform well at sending rates up to 80 msg/s. At 100 msg/s UB still shows low median delivery times but their dispersion is high, because the protocol is getting saturated.

The TR and TR_{prio} yield better performance numbers, even at 100 msg/s. At 10 msg/s the mean is slightly higher than the expected although in these cases, the protocols are not saturated. When the sending rate is low, it may happen that the node which receives a token does not have any message to broadcast. In this case, it simply forwards the token to the next node in the ring. If a message is then broadcast by the application in the first node, then it will have to wait until the token arrives again to that node, thus increasing the delivery time of that particular message and also the mean delivery time. As this happens only to some messages, the delivery time of the rest of the messages is low (due to the low sending rate and the low contention accessing to the network). At higher sending rates this problem no longer arises. At 100 msg/s the dispersion in TR_{prio} is slightly higher as a side effect of the prioritization mechanism, as in UB_{prio}.

Regarding the CH and CH_{prio}, we can see that at low sending rates, the delivery time is high but it decreases noticeably as the sending rate is increased. The design of the CH protocol forces an unordered message received by a node to wait until messages are received from the other nodes. Then, the order is locally (and deterministically) decided without any other message exchange. As the sending rate is increased, messages are forced to wait less time thus reducing the global mean and median delivery time. On the other hand, we can see that the dispersion is kept low in all the cases and more or less similar regardless of the sending rate. The reason of the delay experienced by the messages is mainly because ensuring the causal property imposes a delay on each message significantly greater than the delay imposed by the prioritization mechanism. As the delay imposed by the causal ordering is similar for all the messages sent at a given sending rate, the dispersion of the delivery times is kept low.

In Table 1 we can see that, in a system with 4 nodes, at sending rates up to 60 msg/s, the mean delivery time of any original (non prioritized) protocol is practically equal to the mean delivery time for the corresponding prioritized protocol, which means that the prioritization mechanisms are not imposing any overhead. Something similar happens to the median delivery times. Above 60 msg/s the numbers diverge because the load starts to be too high and then the response depends on each particular protocol.

In a system with 8 nodes, we can see that UB_{prio}, TR_{prio} and CH_{prio} offer good median delivery times at sending rates up to 40 msg/s. Moreover, this numbers are comparable to the ones for their corresponding original (non prioritized versions). At sending rates above 60 msg/s, all the protocols get saturated, in varying degrees, and the delivery times start to get unpractical.

Moreover, we can compare the results from a 4 node system and an 8 node system. In general, we can say that the system scales well when the sending rate is not very high (around 40 msgs/s, as stated above). The UB and UB_{prio} protocols are the ones that offer the worst scalability due its centralized nature. The token-ring and communication history protocol families seem to stand better the increase in the number of nodes.

We can also analyze the dispersion in the values and the effect the prioritization techniques have on it. For instance, consider the results of the 4 node system, at 100 msg/s. Regarding the UB protocol, the difference among the mean delivery time (134.25 ms) and the median (1.28 ms) and third quartile (1.79 ms) shows that there is a significant dispersion in the series of values, caused by the high sending rate. The high dispersion found in UB_{prio} is caused by the high sending rate and by the prioritization mechanism itself. Some messages are delivered very quickly and some other (those with lower priorities) are forced to wait for a while, thus increasing the dispersion in the series of delivery times.

Something similar happens with TR and TR_{prio} at 100 msg/s. The dispersion between the mean and the median with TR_{prio} is bigger than the corresponding with TR. Moreover, in this case, the values of the median, first quartile and third quartile are almost the same in both protocols, but the difference between the mean values is bigger than the one observed with the sequencer-based protocols, which means that the prioritization mechanism used in TR_{prio} yields, as a side effect, a bigger dispersion in the series of delivery times than the one got with the prioritization mechanism used in UB_{prio}. This conclusion can also be drawn observing the results with the 8 node system, at 60 and even at 80 msg/s.

Nevertheless, this effect is less significant in the CH and CH_{prio}. In a system with 4 nodes, they do not show a high dispersion. In a system with 8 nodes, the dispersion is moderated at sending rates up to 60 msg/s. At higher sending rates, the dispersion increases although the increment is lower than that showed

# nodes	msg/s	UB	UB_{prio}	TR	TR_{prio}	CH	CH_{prio}
4	10	4115.69	13898.33	4477.71	5297.12	76773257.38	76738635.95
	40	4034.83	9834.17	4255.10	3504.76	16619838.86	16654534.45
	60	3263.09	9257.53	3515.73	3170.59	10946781.95	10845508.47
	80	3520.73	9834.97	3717.57	2175.87	7666576.22	8495415.76
	100	3376.18	11431.22	3315.10	2030.31	5510957.86	6682313.85
8	10	3741.39	14105.03	4527.08	5180.52	89185553.61	88805093.92
	40	3547.92	12120.56	5129.58	4833.16	21456109.11	21234605.63
	60	3685.69	16840.87	4877.57	3589.27	15693311.47	15131756.34
	80	3644.65	15255.28	4794.20	5024.31	66149667.42	70668213.08
	100	4063.65	15217.12	5750.73	9411.88	112708468.68	93970320.05

Table 2: Mean prioritization times (ns)

by the other protocols.

As a conclusion we can say that the dispersion mainly depends on the length of the queues used by the ordering mechanisms of the protocols. The number of nodes in the system and the sending rate have a direct influence on such lengths.

Regarding the *prioritization times* presented in Table 2, we can analyze the differences among the values of the conventional (non-prioritized) protocols and the prioritized ones. The bigger differences can be found when comparing the UB and the UB_{prio} protocols at any sending rate and with 4 or 8 nodes in the system. At a first glance it seems that the prioritization mechanisms in UB_{prio} is introducing a significant load to the original protocol. Nevertheless, we can see that in all cases, the overhead is around a few microseconds, which compared to the full delivery time (in the order of milliseconds) is negligible.

In the case of the TR and TR_{prio} protocols, the differences are smaller, and again, compared to the full delivery times, are negligible. Moreover, we cannot say that one of the protocols yield better prioritization time numbers than the other in all cases.

Finally, the CH and CH_{prio} protocols deserve a close analysis. On one hand, the absolute values of the prioritization times are several orders of magnitude bigger than those for the other protocols. The reason is that the part of the CH and CH_{prio} protocols considered for taking the measures is basically the code executed to fully handle each incoming message[4]. However, in this case the differences among the values for CH_{prio} and those for CH are negligible by themselves.

Regarding the memory use and the numbers represented in Table 3, we can observe that in general, there are no big differences between the figures for the TR and CH protocols and the ones for their corresponding prioritized versions. Some notable differences exist however, among the numbers of garbage collection runs for the UB and those for UB_{prio}. The reason of these differences is basically the memory overhead suffered by the sequencer node which typically uses more memory than the rest of the nodes of the system[5].

In Appendix A, we depict the evolution of the amount of free memory available for the Java Virtual Machine during each test under different settings. The figures presented in Table 3 can be contrasted against those graphical representations.

[4]For delivering a message to the application, one node must have received at least one message from all nodes in the system, in order to learn about their *logical clocks* and properly order the incoming message respect other messages. This forced pause actually imposes a significant delay, comparable in order of magnitude to the delivery delay itself.

[5]As stated in Section 3.2, these mean numbers are got from the numbers for all the nodes in the system, including its sequencer in case of the UB and UB_{prio} protocols.

# nodes	msg/s	UB	UB_{prio}	TR	TR_{prio}	CH	CH_{prio}
4	10	43.50	54.25	975.75	985.00	51.00	51.00
	40	27.00	30.75	62.00	62.50	41.25	42.00
	60	18.50	22.25	39.25	39.50	20.50	19.75
	80	17.25	26.25	31.50	30.75	17.00	15.75
	100	17.50	22.75	23.75	23.50	17.00	15.75
8	10	35.50	41.62	228.00	230.25	52.75	53.00
	40	18.62	25.12	25.00	24.75	18.62	18.87
	60	19.87	24.87	21.00	21.75	17.37	18.00
	80	22.12	25.50	19.87	20.12	18.00	18.37
	100	23.00	27.00	19.00	19.87	18.00	18.12

Table 3: Mean numbers of garbage collection runs

3.7 Final summary

To summarize the experimental study, we can compare the performance of the conventional protocols and relate it with the performance and overhead of the corresponding prioritized protocols. In this section, we are using global sending rates instead of per-node sending rates; i.e., we are considering how many messages per second have been actually sent by all system nodes.

The UB protocol yields very good performance numbers when the global sending rate is not very high (up to 320 msg/s). When the volume of concurrent messages is too high (due to a high sending rate or because there are too many nodes concurrently broadcasting messages), then the protocol starts to get saturated, and the performance decreases. The main reason of this saturation problem is that the sequencer (which also acts as a regular node that broadcasts and delivers messages) suffers a very high overhead and it is not able to manage and sequence a very high number of concurrent messages.

The TR protocol also shows very good performance results, with sending rates up to 480 msg/s, i.e. it is able to scale better than the UB and UB_{prio} protocols.

The CH protocol does not perform as the other protocols but is the one that best handles high loads (even 800 msg/s). The reason of the lower performance is that this protocol also offers causal delivery guarantees and some messages are forced to wait until causally precedent messages are delivered, thus increasing the mean (and median) delivery time.

Moreover, we have studied the *prioritization time* and can conclude that no overhead is imposed by the prioritization mechanisms. Regarding the memory use, we can conclude that the prioritization mechanisms used in the TR and CH protocols are not imposing any memory overhead (when compared with the TR_{prio} and CH_{prio} protocols, respectively). On the other hand, the results show that the UB_{prio} protocol needs an additional amount of memory, compared with the original non-prioritized UB protocol.

To sum up, we can say that, in general, performance results are similar to those got with the original non-prioritized protocols. In other words, we can confirm that the prioritization techniques presented and tested are not imposing a significant overhead on the original protocols.

4 Conclusions

In this work we continue our study of priority management in total order broadcast protocols we started with a proposal of several techniques to add priority management to total order protocols [11] and a study of how those techniques can be applied in a realistic application and their effectiveness [10].

We have presented an experimental study in which we show that the prioritization techniques do not impose an important overhead (in terms of message delivery latency, processing time and memory use) on the original total order protocols, thus proving that, besides being easy to understand and implement (as shown in [11]) and being useful for replicated database management (as shown in [10]), the techniques are affordable in terms of performance. The main conclusion is that prioritized total order broadcast protocols are a valuable building block that can be used to improve the design and implementation of distributed applications and their performance, as well.

As a second contribution this experimental study can be seen also as a performance comparison among conventional non-prioritized total order protocols. The results of this comparison show that sequencer-based and privilege-based protocols offer a comparable performance when the number of nodes is small (4 or 8) and the sending rate is not too high (around 60 msg/s). As the number of nodes or the sending rate is increased the sequencer-based protocols start to get saturated and the communication history improve their performance. At higher sending rates communication history protocols are the ones that can stand the load, although the performance is not the best.

A Graphic results

In this appendix we show some graphic results obtained from the tests we have performed. In Section A.1 we present some results related to the prioritization times while Section A.2 analyses the memory use.

A.1 Prioritization time analysis

To evaluate the cost of the prioritization mechanisms applied in the UB_{prio}, TR_{prio} and CH_{prio} protocols we have recorded the prioritization time of each message, as explained in Section 3.4. These values can be used to get an idea of how expensive are the prioritization mechanisms. The values are grouped in sets of 1000 values. The mean value of each set is computed and then depicted in a single curve. Each curve plotted represents the evolution of the prioritization times of a given test. To ease the comparison, related curves are plotted in the same figure.

In Section A.3 we show the graphic representation of the prioritization times, in systems with 4 and 8 nodes, with different conventional and prioritized total order protocols, at different sending rates.

We can see in Figures 1 and 2 that the prioritization times for the tests with the UB_{prio} protocols are higher (worse) than the corresponding tests with UB. For instance, in Figure 1 the prioritization times of the tests with UB_{prio} are more than twice the corresponding times with UB. Nevertheless, as explained in Section 3.6, by comparing these times against the corresponding delivery times we can see that the extra cost observed in the tests with UB_{prio} is negligible.

Regarding the token-based protocols, we can see in Figures 3 and 4 that the TR_{prio} protocol needs more time than the TR protocol to manage the first messages but it needs less time to manage the *final* messages. These results suggest that in a system with a constant sending rate, the prioritized TR_{prio} protocol behaves even better than the original TR protocol. Nevertheless, again the differences among the prioritization times for the TR and the TR_{prio} protocols are negligible when compared against the corresponding delivery times (as shown in Table 1).

Finally, Figures 5 and 6 show that there are no significant differences among the results for the conventional CH protocol and the prioritized CH_{prio} protocol, in systems with 4 and 8 nodes, respectively.

A.2 Memory use analysis

To evaluate the cost of the prioritization mechanisms in terms of memory use, during each test we have recorded the values of three indicators: the total, free and maximum amount of memory available to the Java Virtual Machine. During a given test, each time a message is received by the test application, the values of these indicators are recorded. The figures in Sections A.4 and A.5 show the evolution of these indicators for different nodes in different settings, with 4 and 8 nodes, respectively.

In the case of the UB and UB_{prio} protocols, we show two figures for each test, one for the sequencer node (usually the first node) and another for a non-sequencer node (typically the second node). As shown in the figures, a sequencer node has different memory requirements than the rest of the nodes. For the rest of the protocols, we only show the figure corresponding to a node (typically the first one).

A.2.1 System with 4 nodes

In Figures 7 to 46, we show the evolution of the memory use indicators in a system with 4 nodes, with a sending rate ranging from 10 msg/s to 100 msg/s, with different protocols, in different nodes.

For instance, in Figures 7 and 8 we show the results for the UB protocol in the first node (the sequencer) and the second. At a first glance, there are some significant differences between both figures. The most important is related with the *free memory* curve. In Figure 7, the curve shows a cyclic behavior. We have contrasted these results against those obtained from the Java garbage collector, which allows us to draw some conclusions. During a short period of time, the amount of free memory deeply decreases. After a *minor garbage collection* some memory is recovered and the amount of free memory increases. In such a minor garbage collection, (typically small) short-lived objects are collected. This cycle is repeated many times, until the Java Virtual Machine is close to run out of memory. Then, a *major garbage collection* is ran. This collection is able to recover a high amount of memory, corresponding to long-lived objects that are no longer used. After such a collection, this cyclical behavior is repeated.

In Figure 8 we show the corresponding curve for the second node in the system. The behavior is, in essence, similar to the one of the first node. An important difference is that the garbage collections are not so frequent than in Figure 7. The main reason is that the first node is the sequencer and needs to allocate and use much more memory than the rest of the nodes. For the same reason, the amount of free memory available to the Java Virtual Machine in the second node decreases more slowly than in Figure 7.

This behavior is also found in Figures 15 and 16 (40 msg/s), 23 and 24 (60 msg/s), 31 and 32 (80 msg/s) and 39 and 40 (100 msg/s). These figures have small differences regarding Figures 7 and 8, due to the higher sending rates. The most notable difference is the length of the *steps* (as the sending rate is increased, the amount of memory released by the garbage collector in each collection is slightly bigger). Moreover, the *total memory* curve shows that increasing the sending rate causes some (light) increase in the total amount of memory managed by the Java Virtual Machine.

These results can be compared against the corresponding for the UB_{prio} protocols, in Figures 9 and 10 (10 msg/s), 17 and 18 (40 msg/s), 25 and 26 (60 msg/s), 33 and 34 (80 msg/s) and 41 and 42 (100 msg/s). Although particular differences can be found, we can say that, in general, the behavior observed in a system with a conventional (non prioritized) protocol under a given setting is similar to that of the corresponding system with its prioritized protocol version. There are, nevertheless some differences regarding the number of garbage collections performed in a system running the UB protocol and the corresponding system with UB_{prio}, as pointed out in Section 3.6. We can see that in the tests with UB_{prio} the number of garbage collections is higher than in the corresponding tests with UB, which means that UB_{prio}, under a given setting, needs more memory than UB under the same setting.

In Figures 11, 19, 27, 35 and 43 we show the behavior of the TR protocol in a 4 node system at a sending rate of 10, 40, 60, 80 and 100 msg/s, respectively. At 10 msg/s (Figure 11), we can see that the system is performing really well. First of all, the *total memory* curve shows a slightly decreasing tendency that finally seems to stabilize. Moreover, the high *density* of the *free memory* curve shows that there are a lot of short-lived objects that are being successfully collected by means of minor garbage collections, which actually means that no heavy major garbage collections are usually needed.

As the sending rate is increased, the behavior gets more similar to the behavior of the sequencer-based protocols. Basically, the garbage collections become less frequent but the amount of memory recovered in each collection is increased. We can also see that the total amount of memory needed by TR also increases (respect to that depicted in Figure 11).

In Figures 12, 20, 28, 36 and 44 we show the behavior of the corresponding prioritized TR_{prio} protocol at 10, 40, 60, 80 and 100 msg/s, respectively. As can be seen, the behavior is very similar to the behavior of the conventional (non-prioritized) TR protocol, which clearly shows that the prioritization mechanism applied in TR_{prio} is not imposing any overhead on the memory use.

In Figures 13, 21, 29, 37 and 45 we show the behavior of the CH protocol in a 4 node system at a sending rate of 10, 40, 60, 80 and 100 msg/s, respectively. Although there are some minor differences, the behavior is basically similar. As the sending rate is increased the frequency of the garbage collections decreases and the amount of memory recovered increases. Nevertheless, the tendency of the *free memory* and *total memory* curves show that the system evolves in a very controlled manner. Figures 14, 22, 30, 38 and 46 show the results for the corresponding settings with the CH_{prio} protocol. As we can see, the results are very similar to the ones from the original CH protocol in the corresponding settings, which again means that no memory overhead is imposed by the prioritization mechanism used by the CH_{prio} protocol.

A.2.2 System with 8 nodes

In Figures 47 to 86, we show the evolution of the memory use indicators in a system with 8 nodes, with a sending rate ranging from 10 msg/s to 100 msg/s, with different protocols, in different nodes.

In general, we can see that the results are similar to those presented in Section A.2.1, for systems with 4 nodes. We can note, nevertheless some differences between a given test in a setting with 4 nodes and the corresponding test with 8 nodes. For instance, the sequencer node of the UB protocol at 10 msg/s shows a significantly different behavior in systems with 4 and 8 nodes, as shown in Figures 7 and 47. In the system with 8 nodes the global load stood by the sequencer is twice the load in the system with 4 nodes. For this reason, the sequencer of the 8 node system tends to run out of memory sooner than the sequencer in the 4 node system. In both cases, the garbage collector seems to work properly, recovering a big amount of unused memory. The behavior of the sequencer in the UB_{prio} protocol, also at 10 msg/s, show a similar difference in systems with 4 and 8 nodes. On the other hand, the behavior of the non-sequencer nodes in the UB and UB_{prio} protocols is also similar in systems with 4 and 8 nodes.

Regarding the token-based protocols, we can see that the TR and TR_{prio} protocols show a similar behavior in both systems, with 4 and 8 nodes. The main differences lie in the frequency with which the garbage collector is activated to perform minor collections.

Something very similar happens to the communication history protocols. In some cases, besides these differences in the garbage collection activation frequency, it is possible to find small differences in the quantity of memory recovered. In any case, we consider that these differences are negligible.

A.3 Prioritization time plots

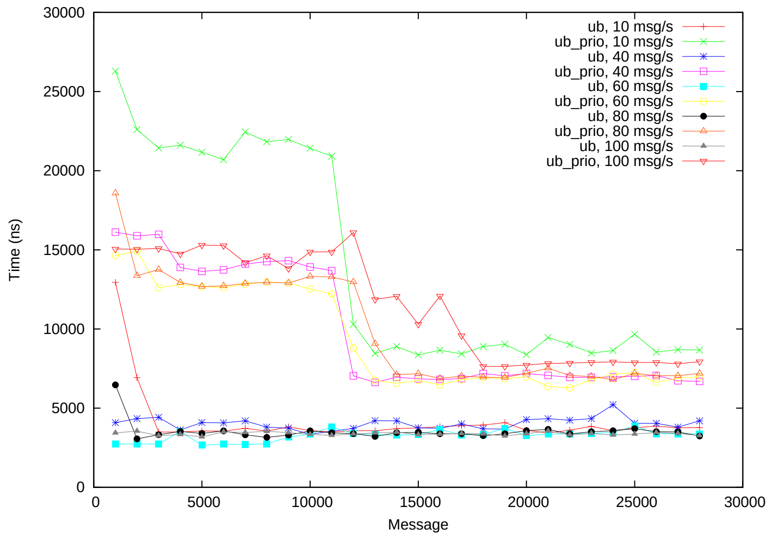

Figure 1: Prioritization times (UB and UB_{prio}), 4 nodes

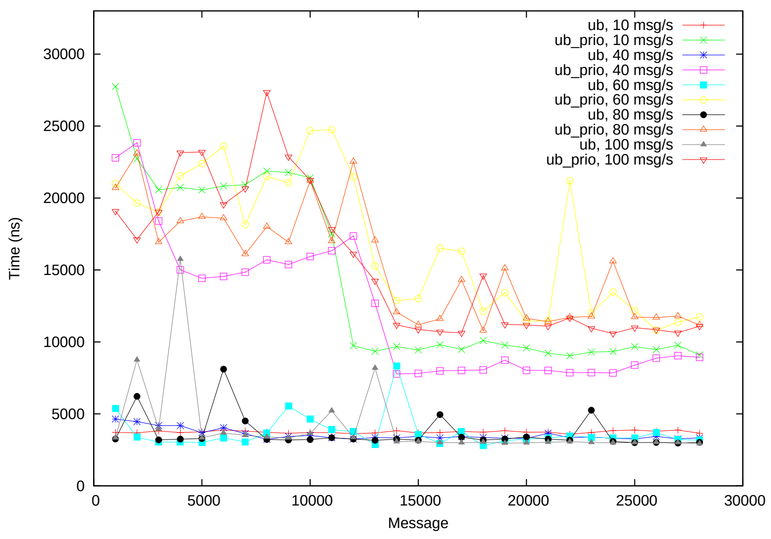

Figure 2: Prioritization times (UB and UB_{prio}), 8 nodes

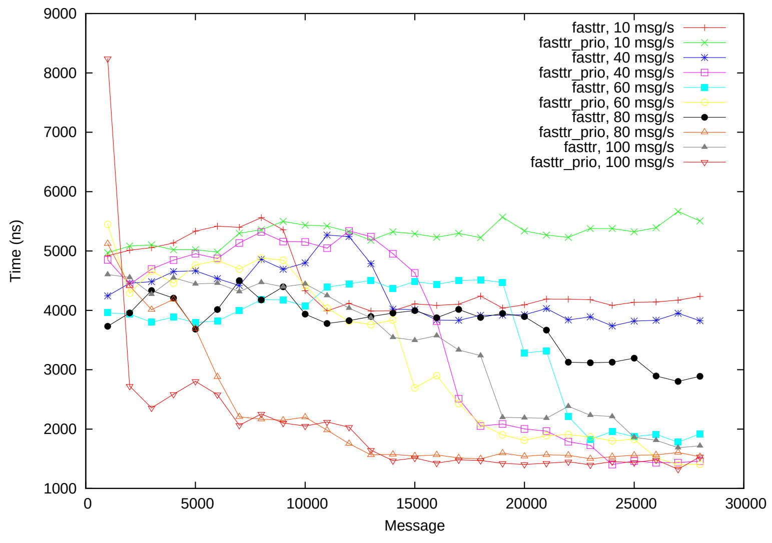

Figure 3: Prioritization times (TR and TR_{prio}), 4 nodes

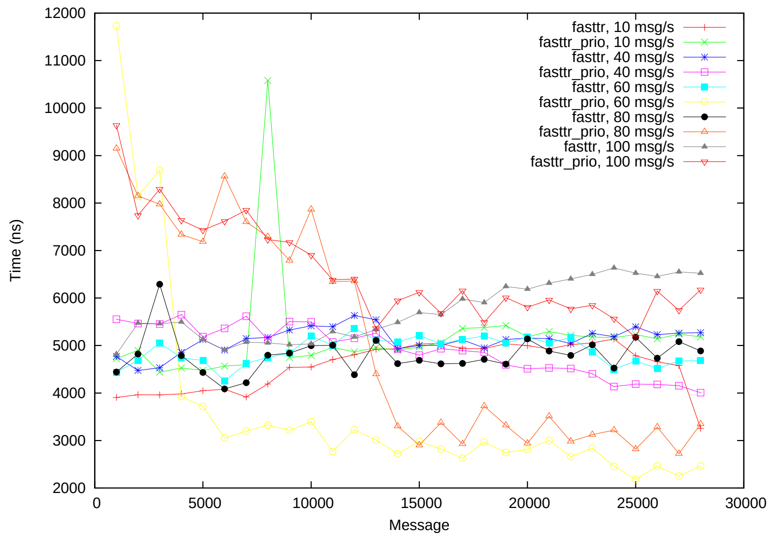

Figure 4: Prioritization times (TR and TR_{prio}), 8 nodes

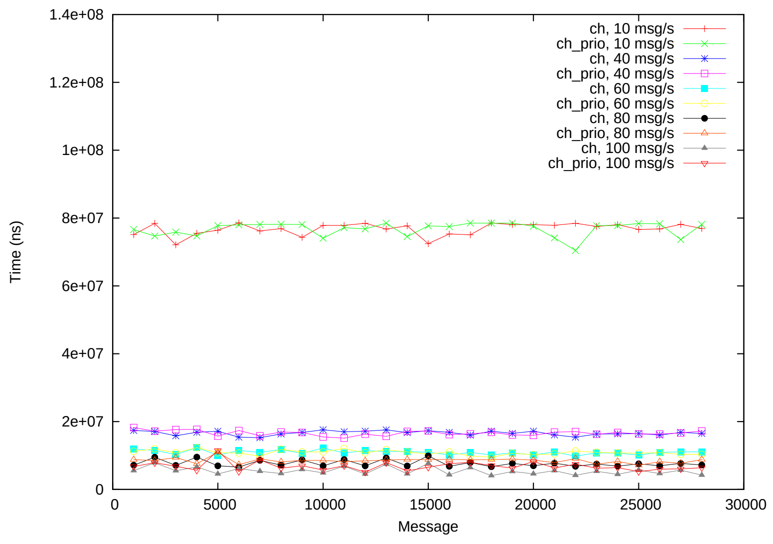

Figure 5: Prioritization times (CH and CH_{prio}), 4 nodes

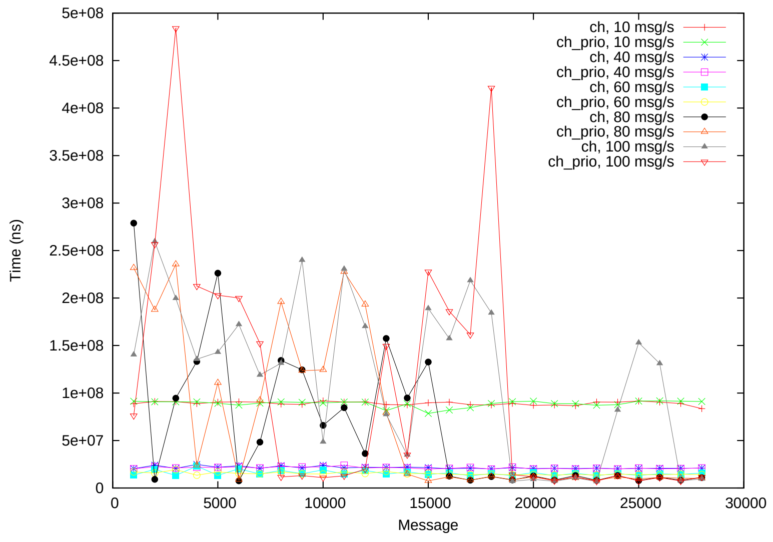

Figure 6: Prioritization times (CH and CH_{prio}), 8 nodes

A.4 Memory use graphic results in a system with 4 nodes

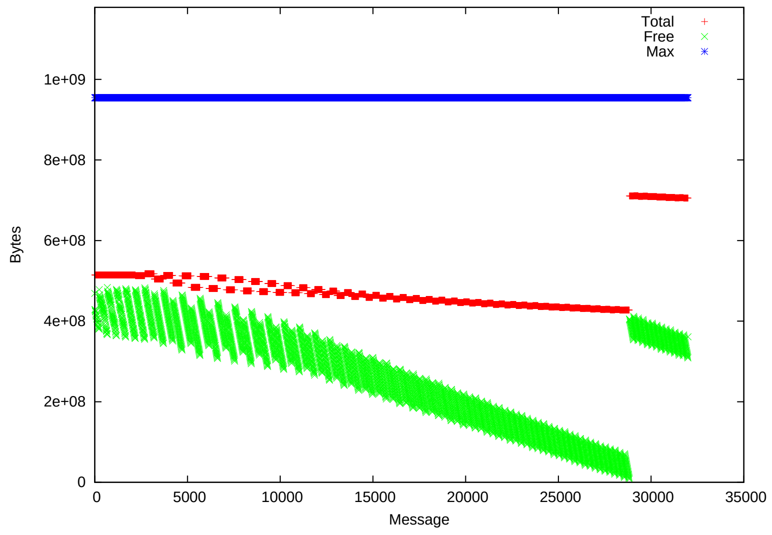

Figure 7: UB, 10 msgs/s, node1 (sequencer)

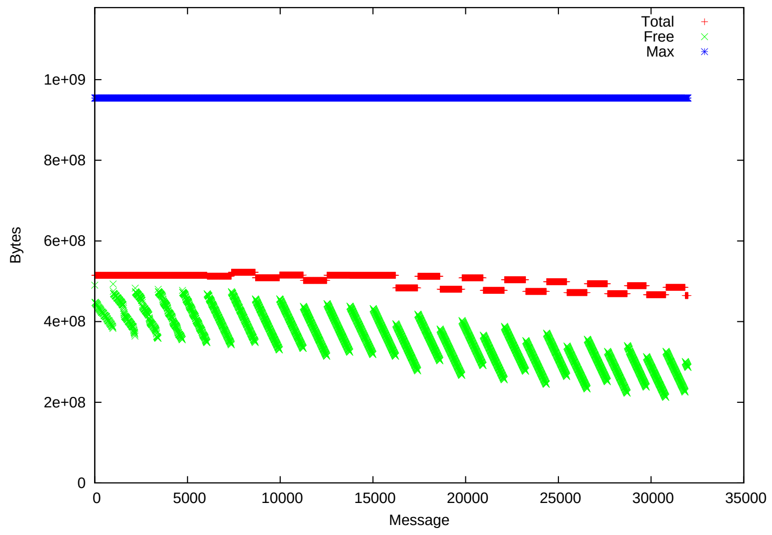

Figure 8: UB, 10 msg/s, node2

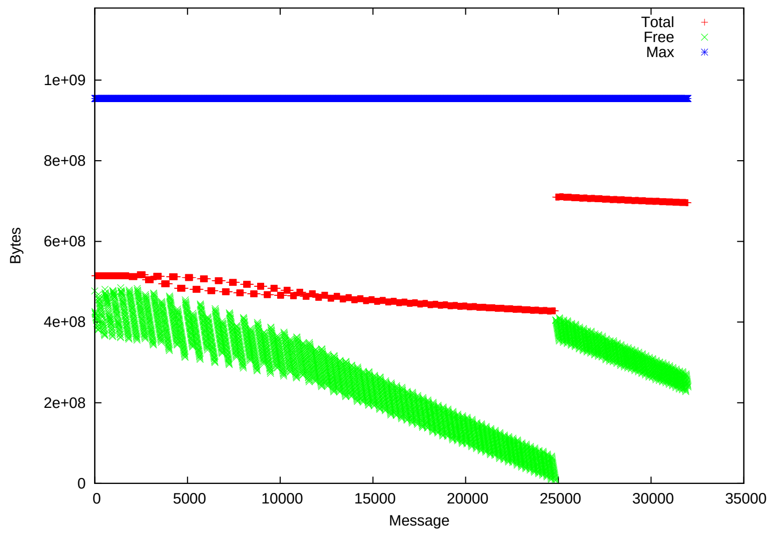

Figure 9: UB_{prio}, 10 msg/s, node1 (sequencer)

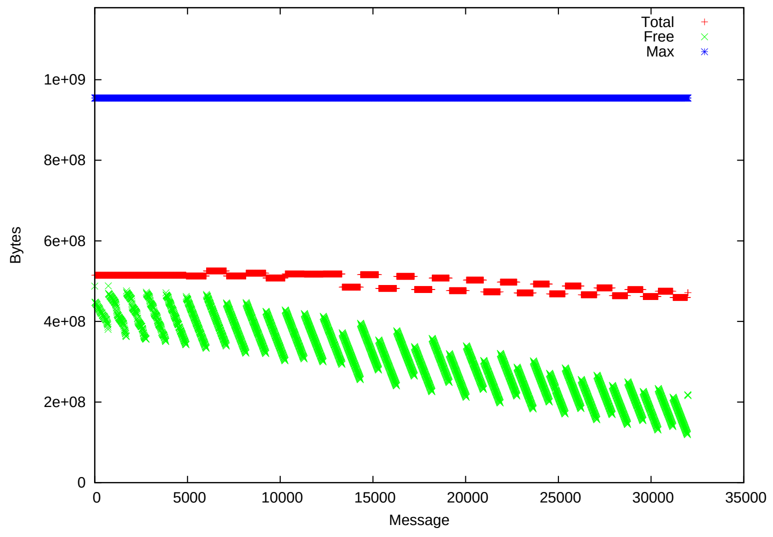

Figure 10: UB_{prio}, 10 msg/s, node2

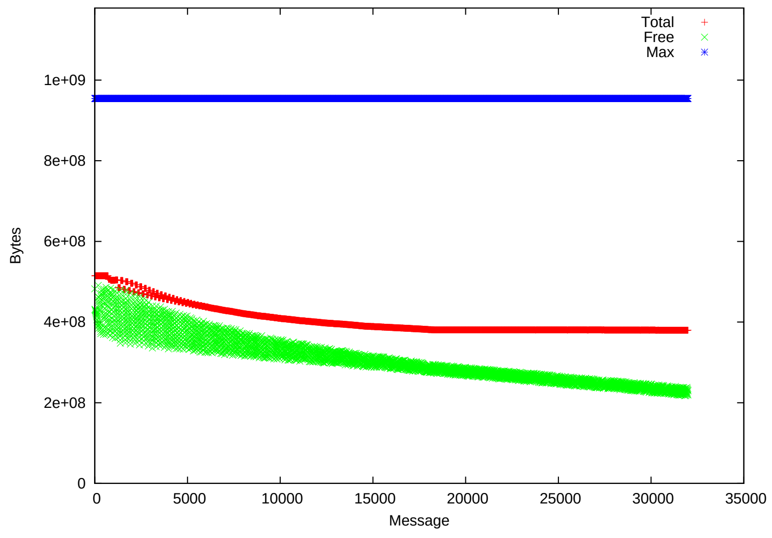

Figure 11: TR, 10 msg/s, node1

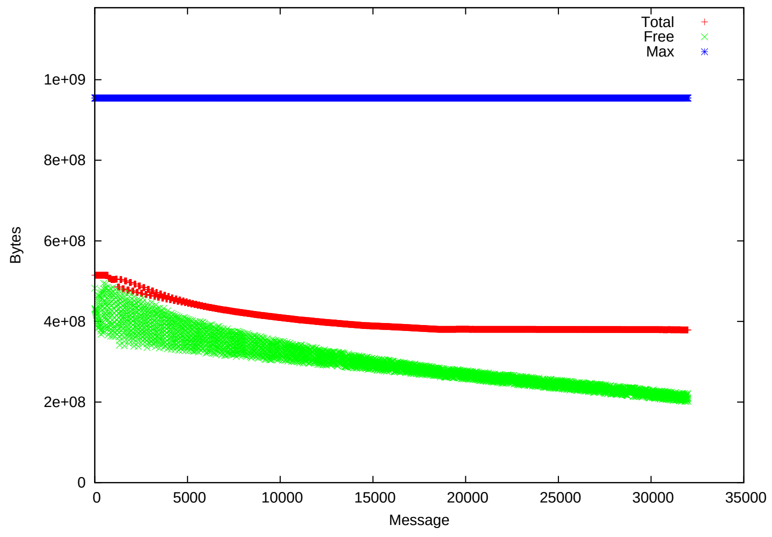

Figure 12: TR_{prio}, 10 msg/s, node1

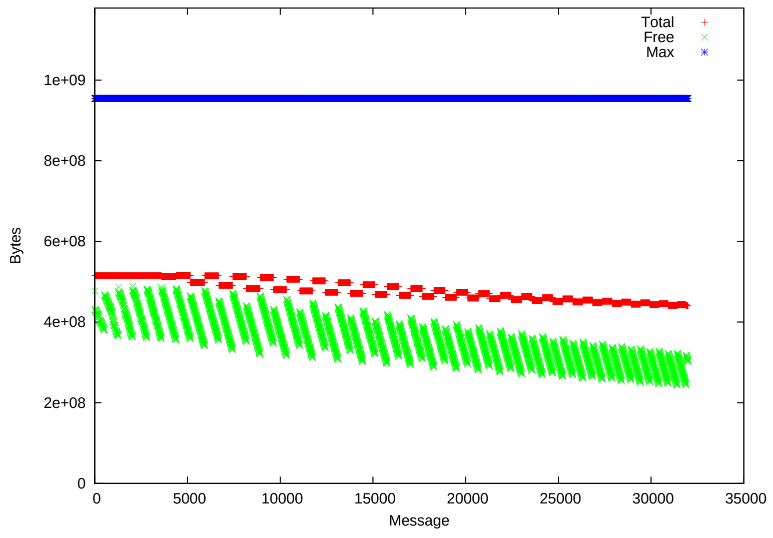

Figure 13: CH, 10 msg/s, node1

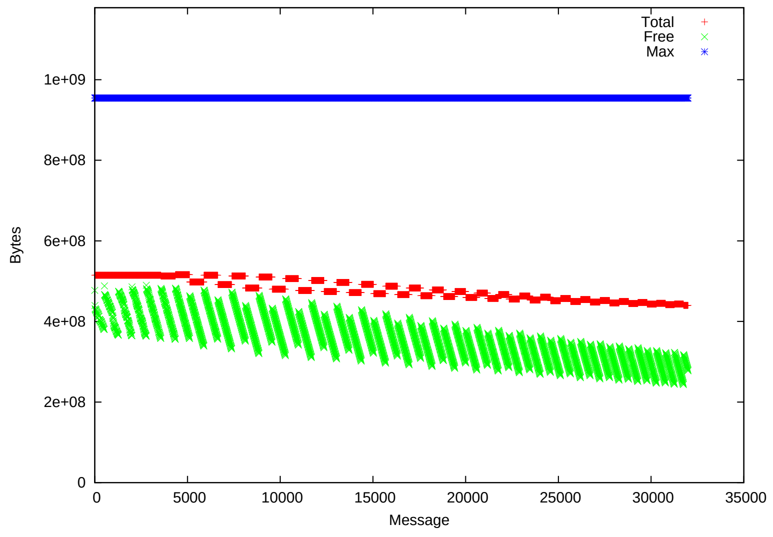

Figure 14: CH_{prio}, 10 msg/s, node1

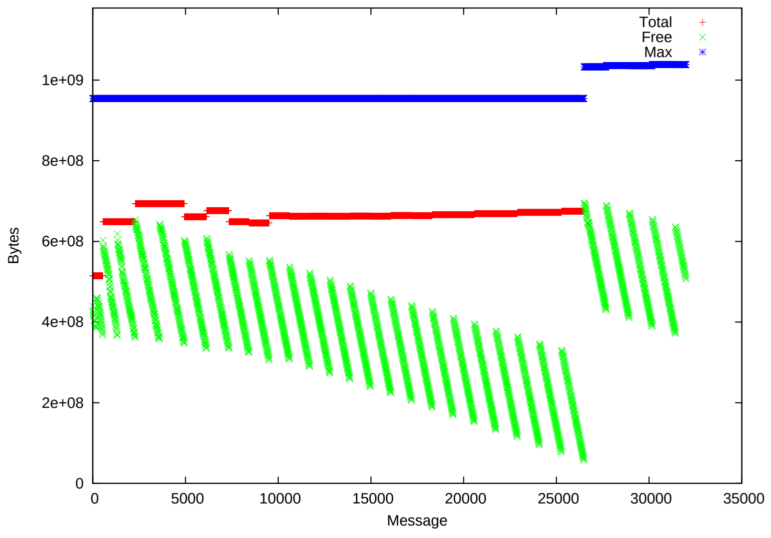

Figure 15: UB, 40 msg/s, node1 (sequencer)

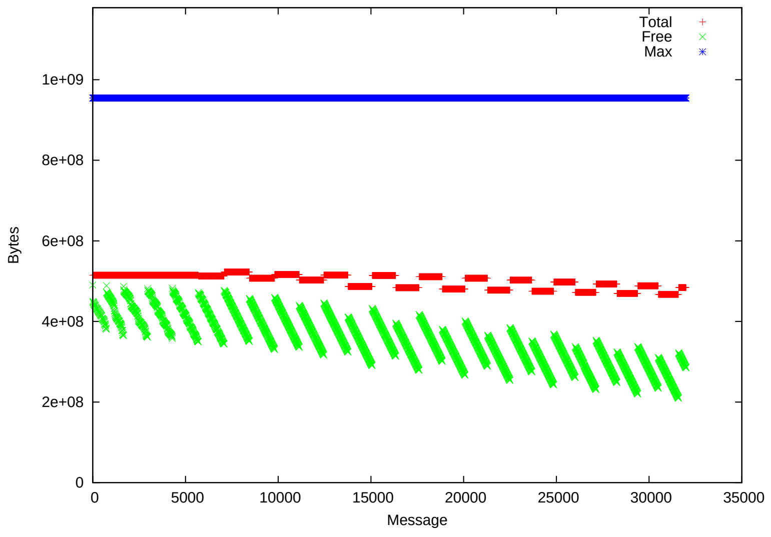

Figure 16: UB, 40 msg/s, node2

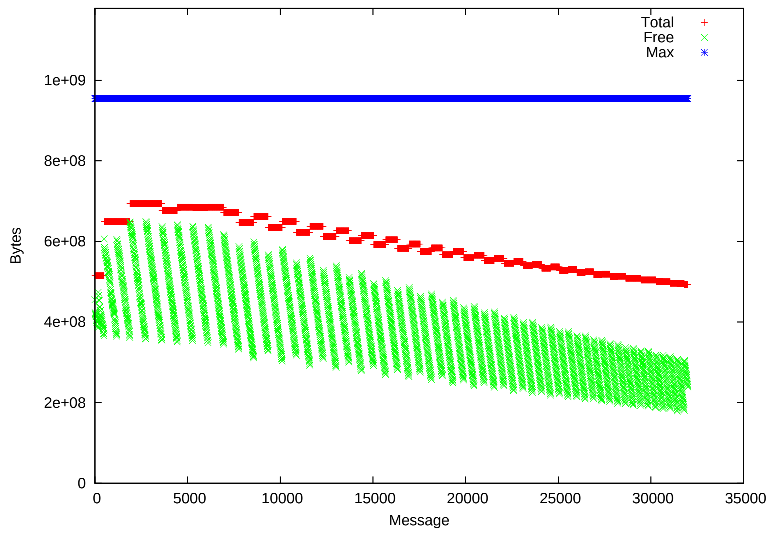

Figure 17: UB_{prio}, 40 msg/s, node1 (sequencer)

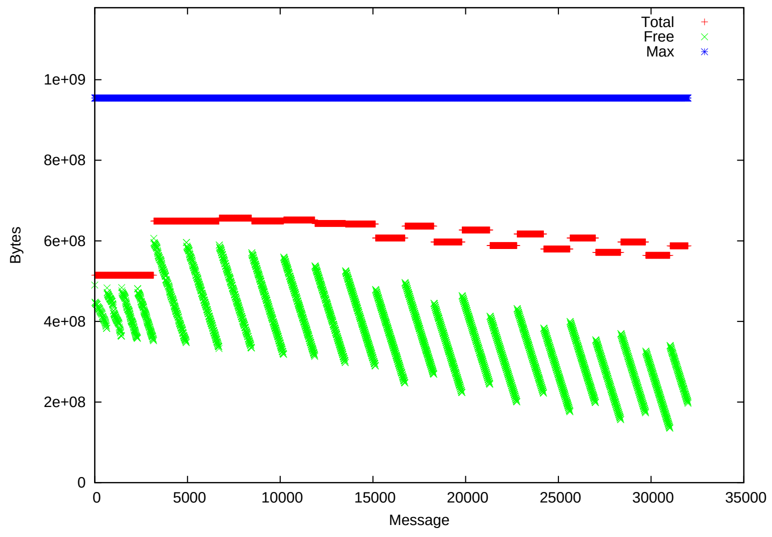

Figure 18: UB_{prio}, 40 msg/s, node2

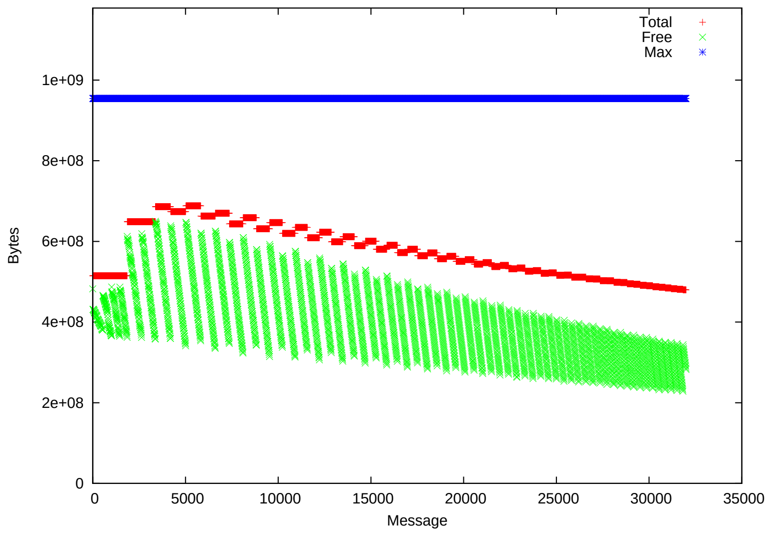

Figure 19: TR, 40 msg/s, node1

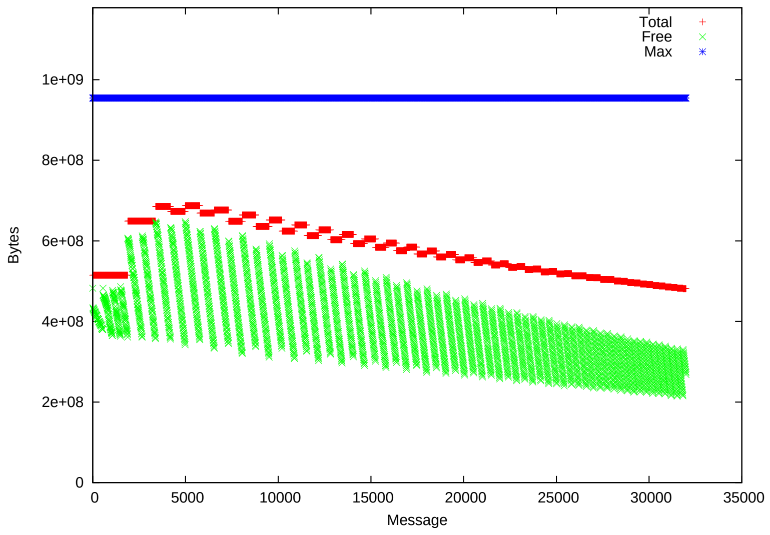

Figure 20: TR_{prio}, 40 msg/s, node1

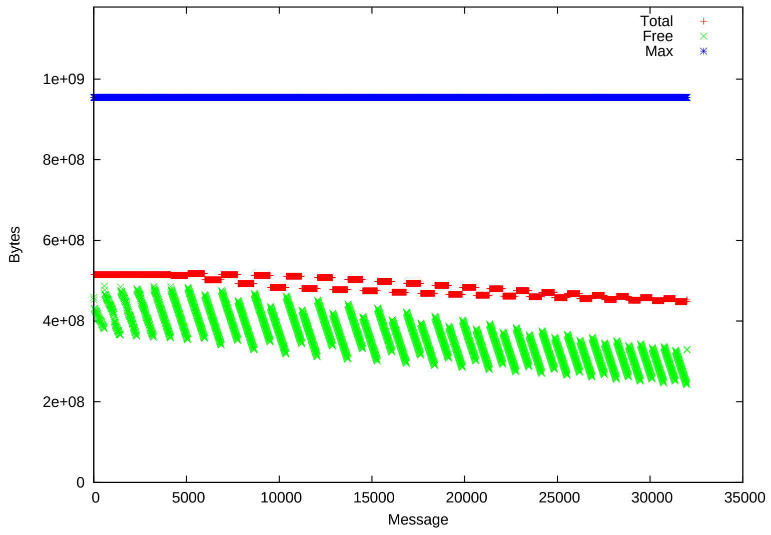

Figure 21: CH, 40 msg/s, node1

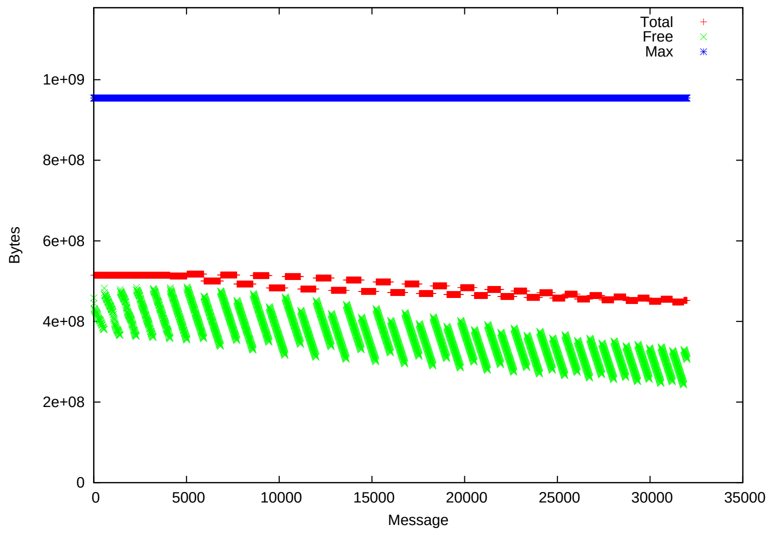

Figure 22: CH_{prio}, 40 msg/s, node1

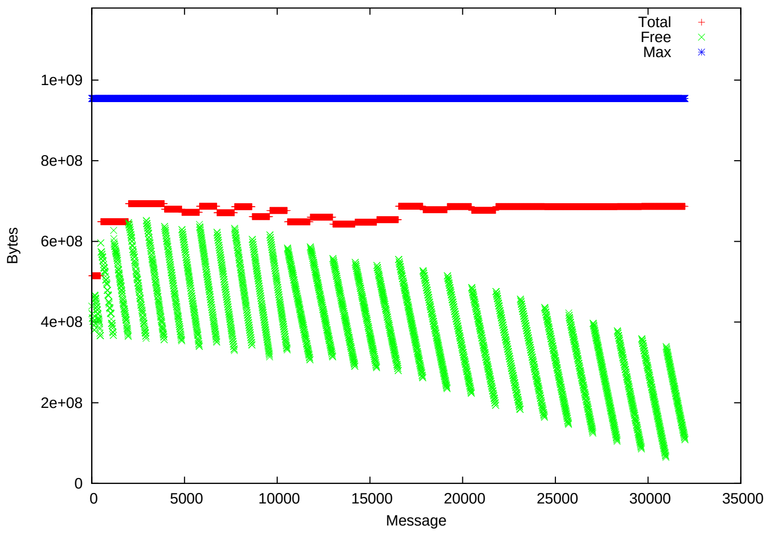

Figure 23: UB, 60 msg/s, node1 (sequencer)

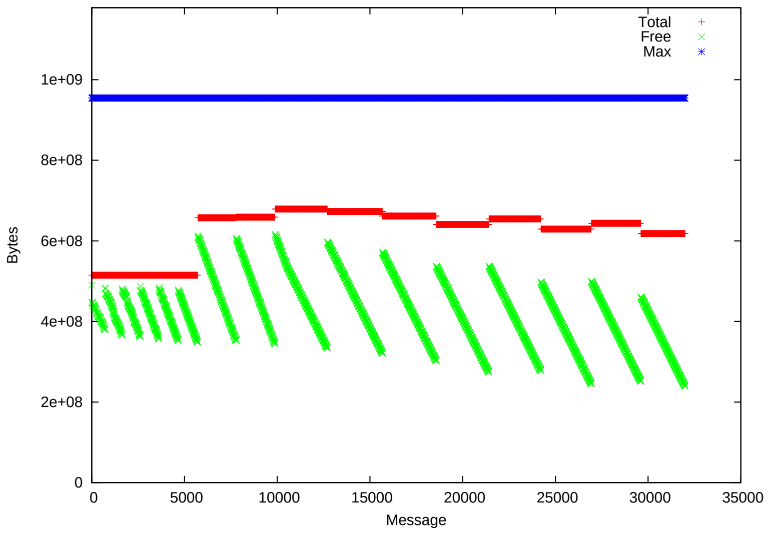

Figure 24: UB, 60 msg/s, node2

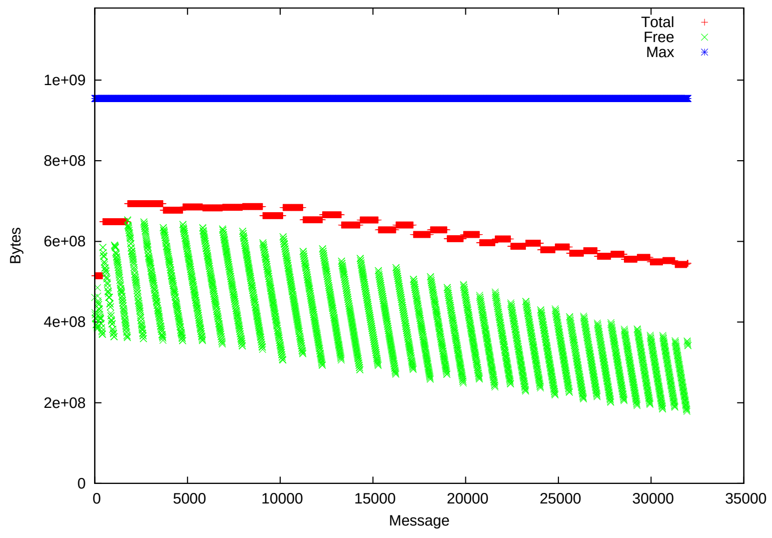

Figure 25: UB_{prio}, 60 msg/s, node1 (sequencer)

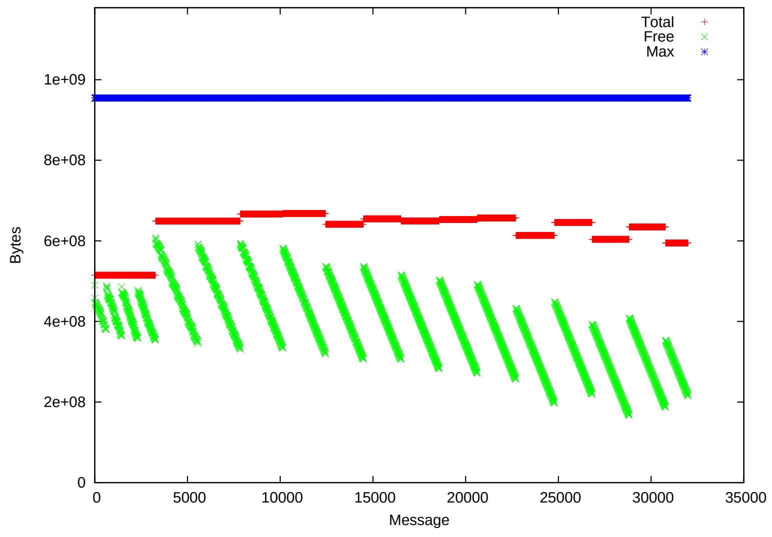

Figure 26: UB_{prio}, 60 msg/s, node2

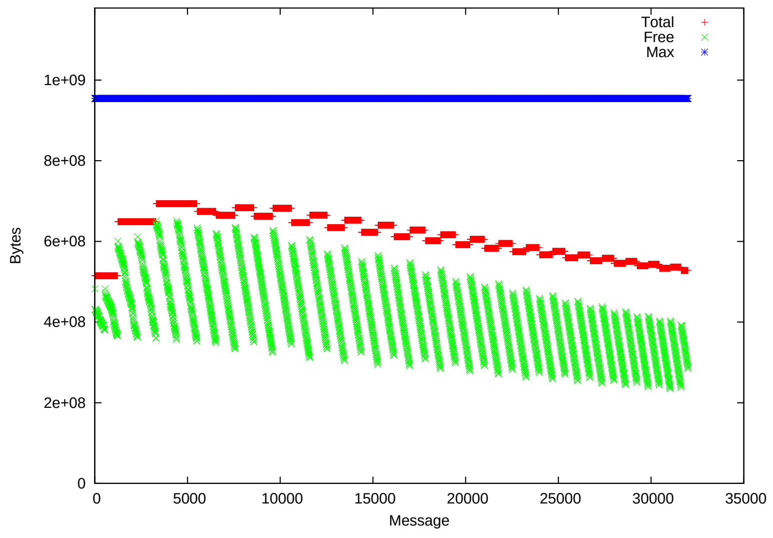

Figure 27: TR, 60 msg/s, node1

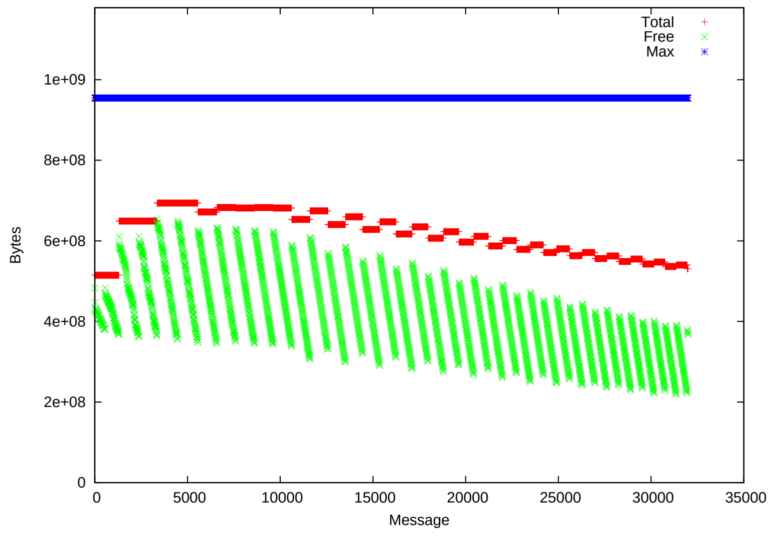

Figure 28: TR_{prio}, 60 msg/s, node1

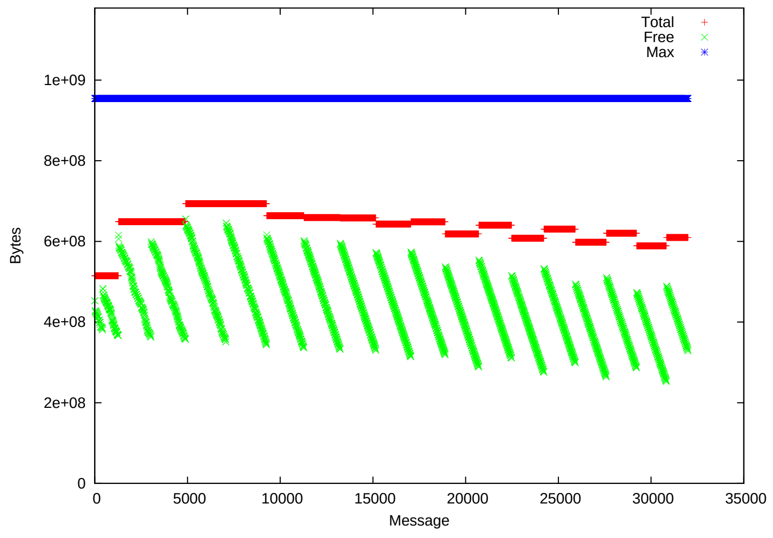

Figure 29: CH, 60 msg/s, node1

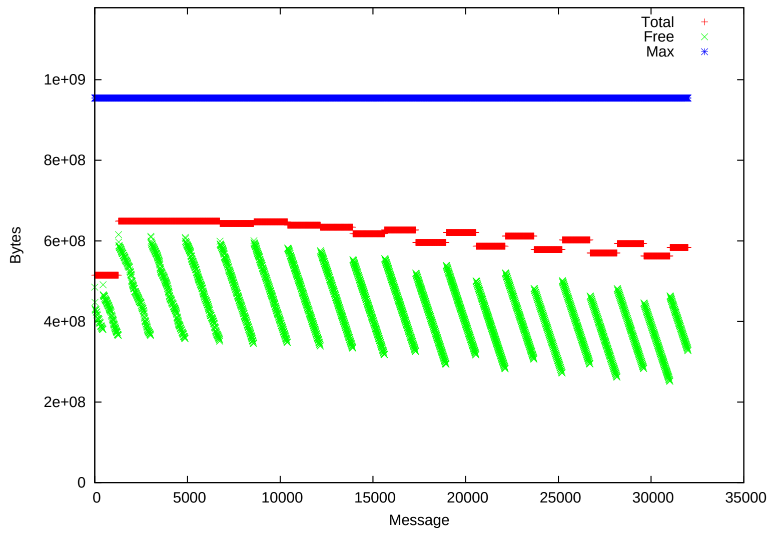

Figure 30: CH_{prio}, 60 msg/s, node1

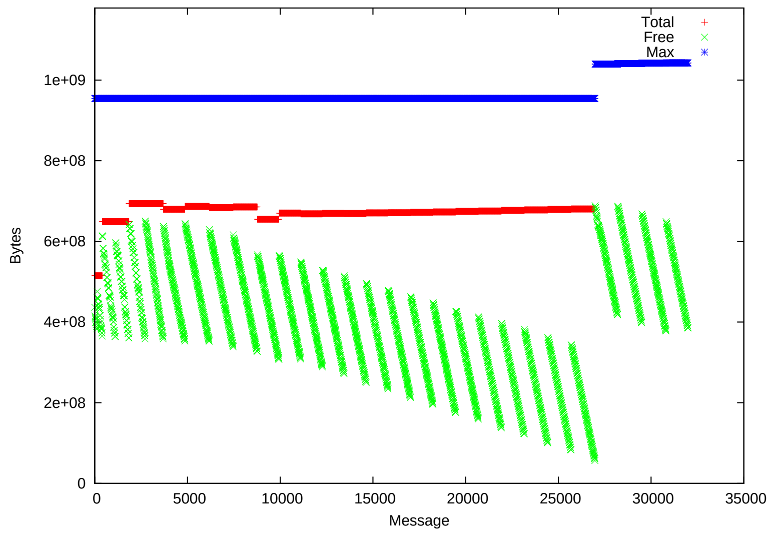

Figure 31: UB, 80 msg/s, node1 (sequencer)

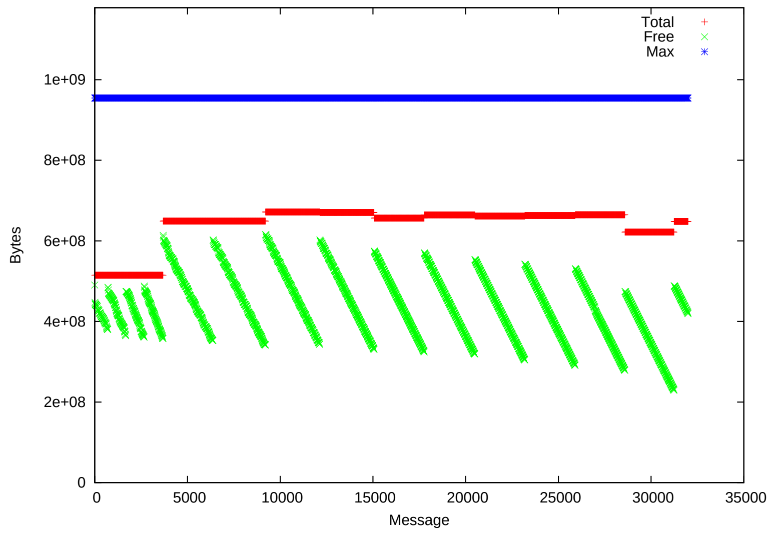

Figure 32: UB, 80 msg/s, node2

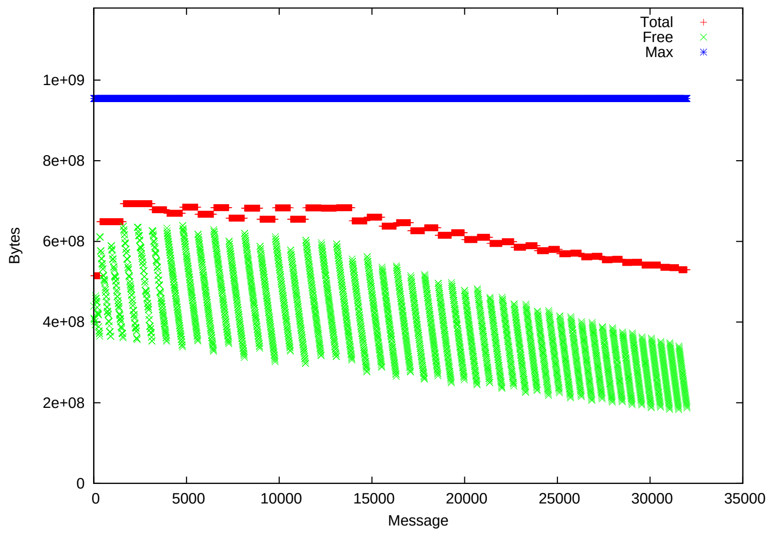

Figure 33: UB_{prio}, 80 msg/s, node1 (sequencer)

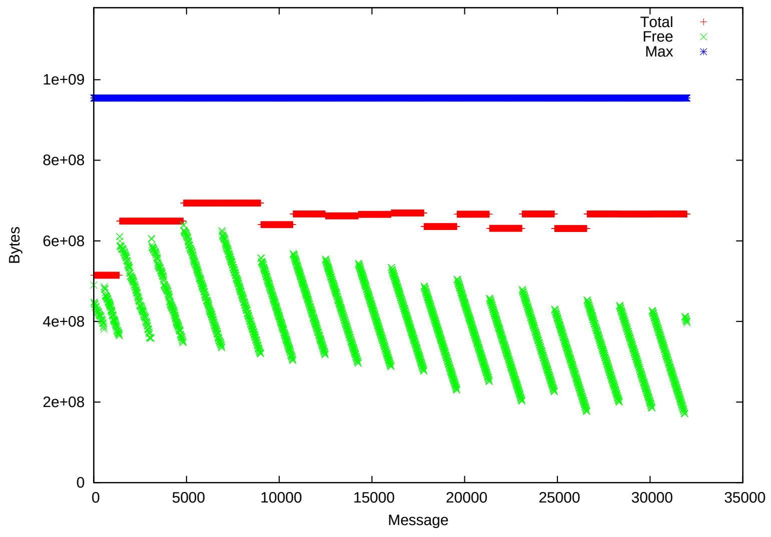

Figure 34: UB_{prio}, 80 msg/s, node2

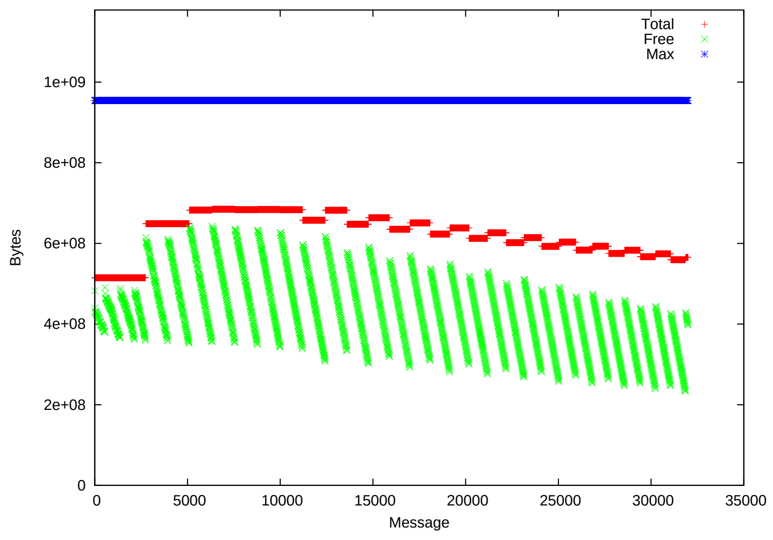

Figure 35: TR, 80 msg/s, node1

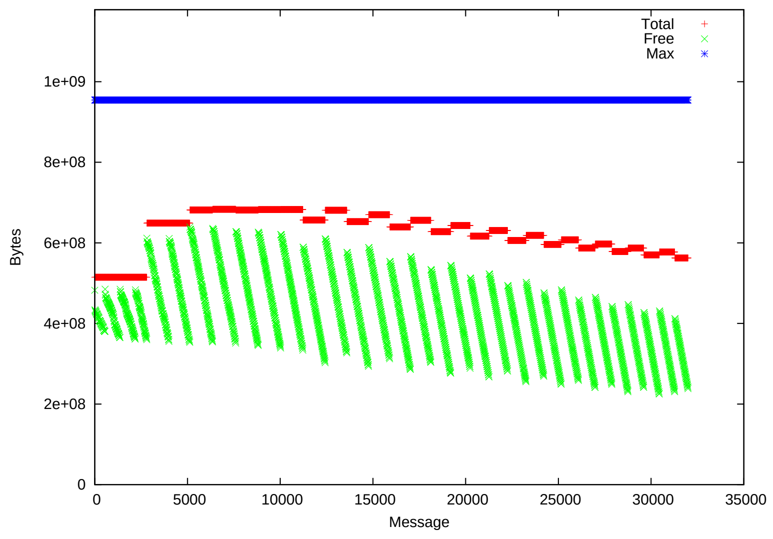

Figure 36: TR_{prio}, 80 msg/s, node1

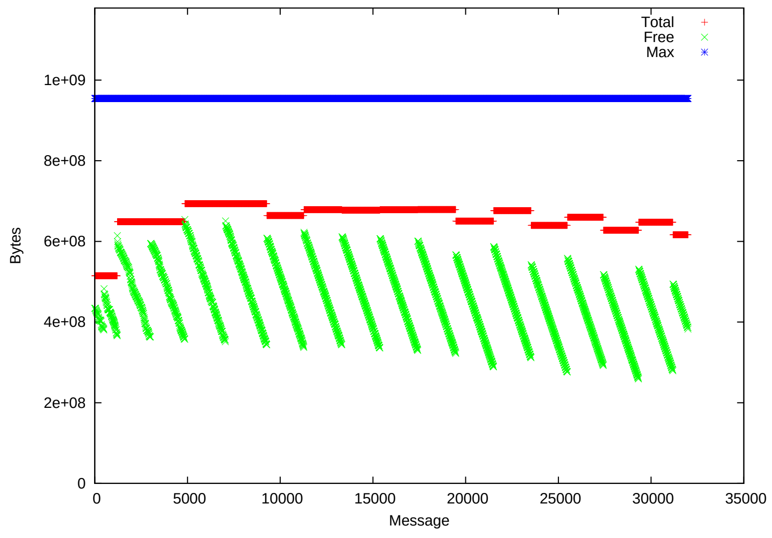

Figure 37: CH, 80 msg/s, node1

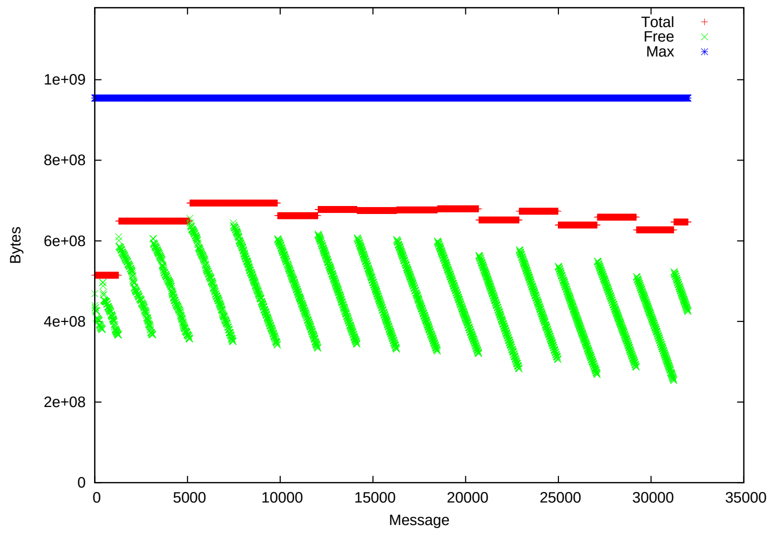

Figure 38: CH_{prio}, 80 msg/s, node1

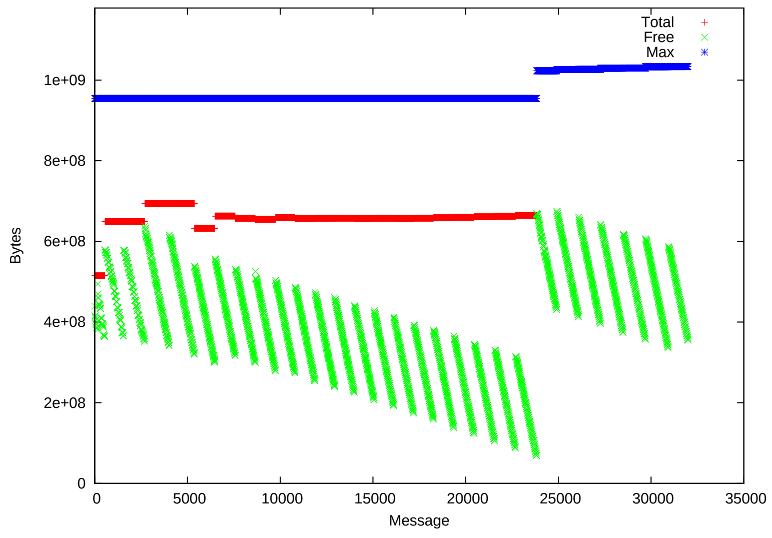

Figure 39: UB, 100 msg/s, node1 (sequencer)

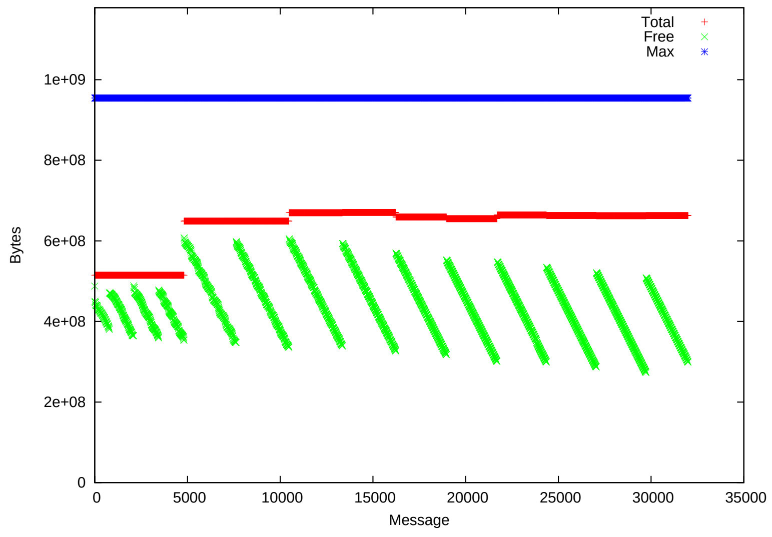

Figure 40: UB, 100 msg/s, node2

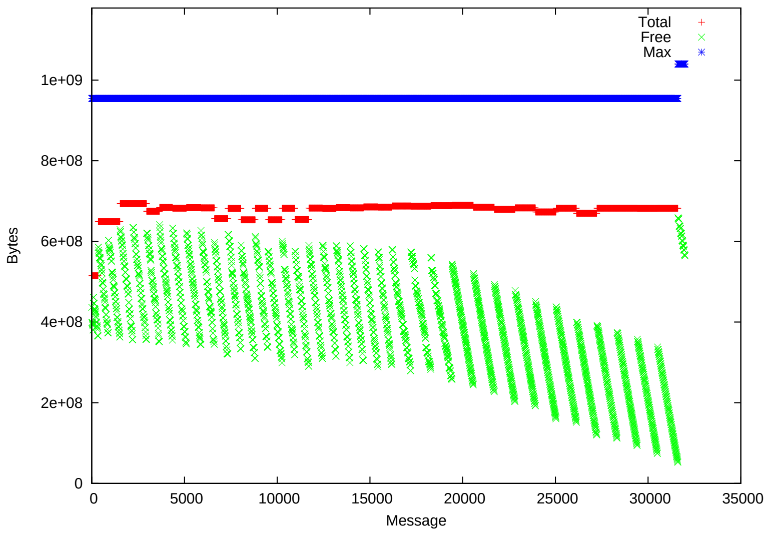

Figure 41: UB_{prio}, 100 msg/s, node1 (sequencer)

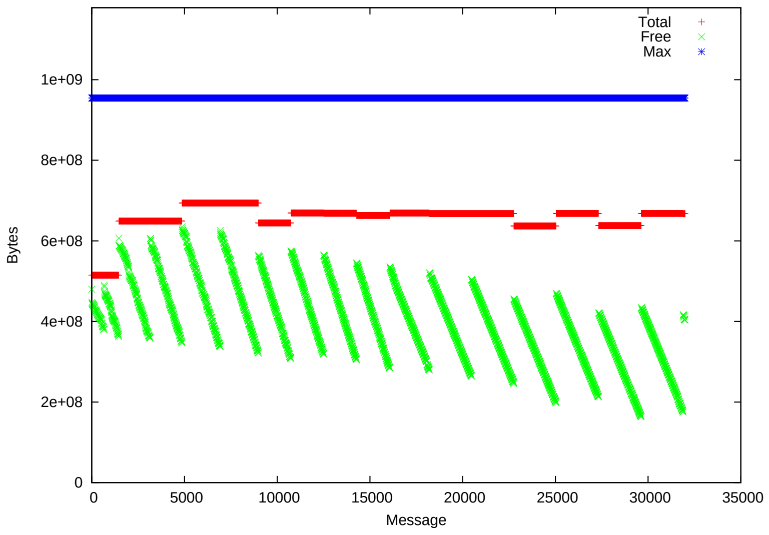

Figure 42: UB_{prio}, 100 msg/s, node2

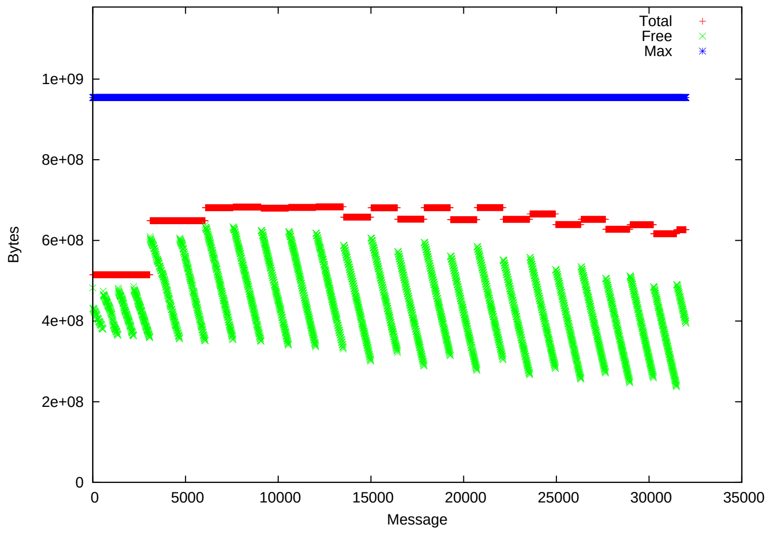

Figure 43: TR, 100 msg/s, node1

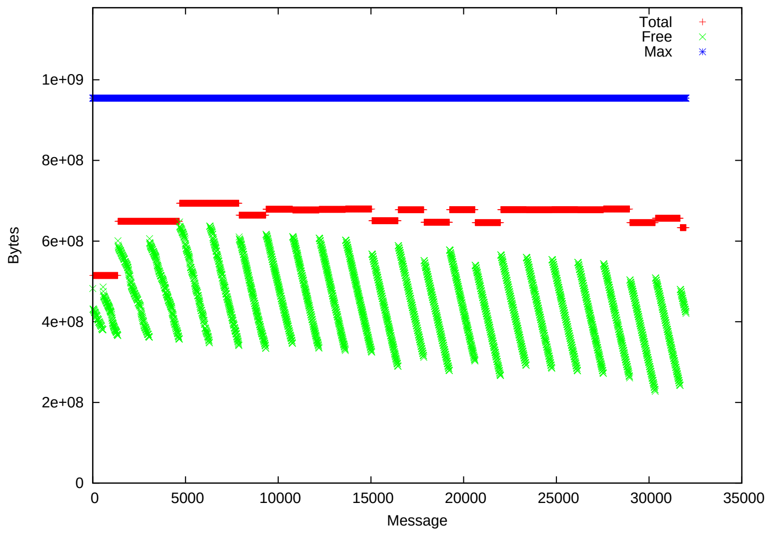

Figure 44: TR_{prio}, 100 msg/s, node1

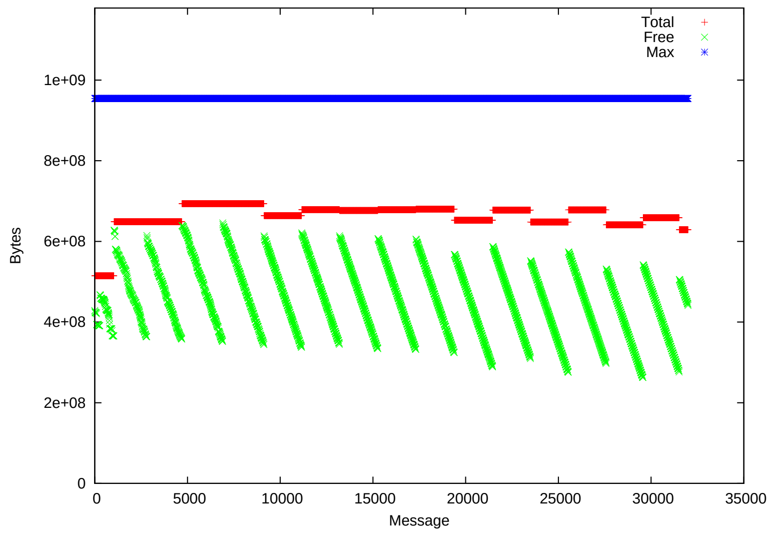

Figure 45: CH, 100 msg/s, node1

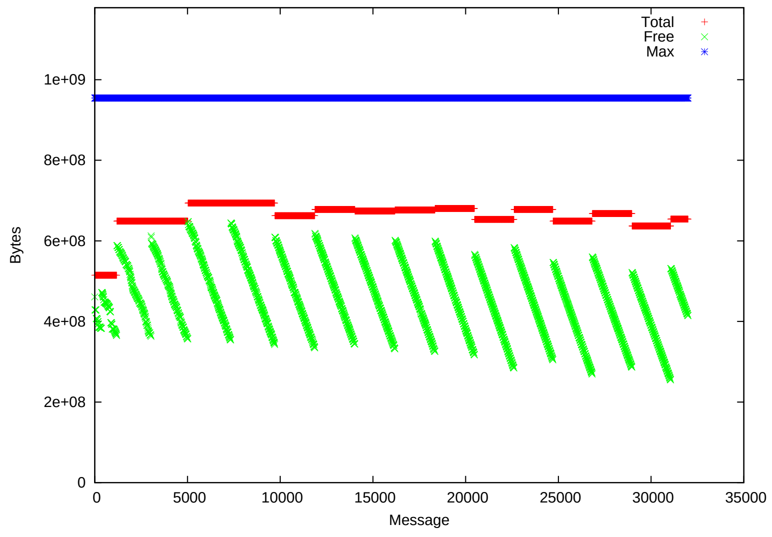

Figure 46: CH_{prio}, 100 msg/s, node1

A.5 Memory use graphic results in a system with 8 nodes

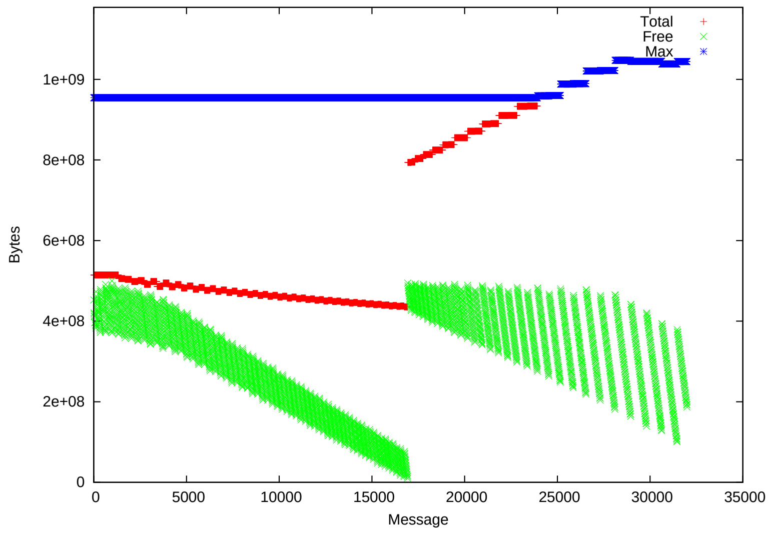

Figure 47: UB, 10 msg/s, node1 (sequencer)

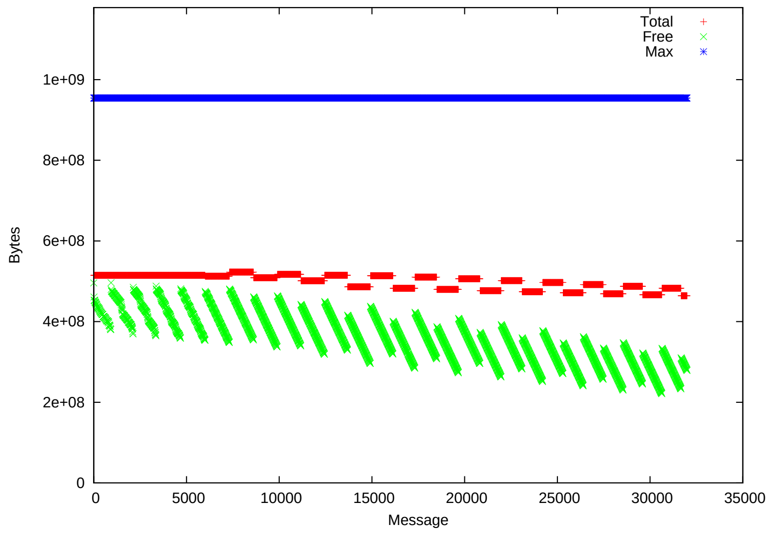

Figure 48: UB, 10 msg/s, node2

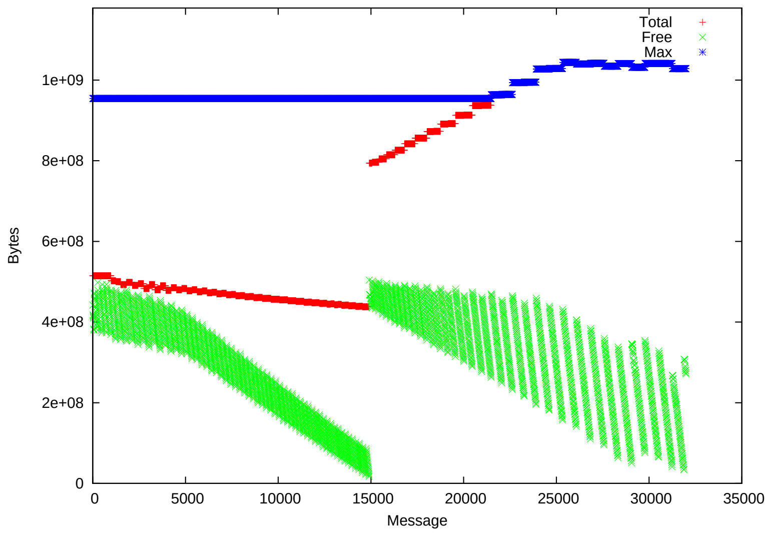

Figure 49: UB_{prio}, 10 msg/s, node1 (sequencer)

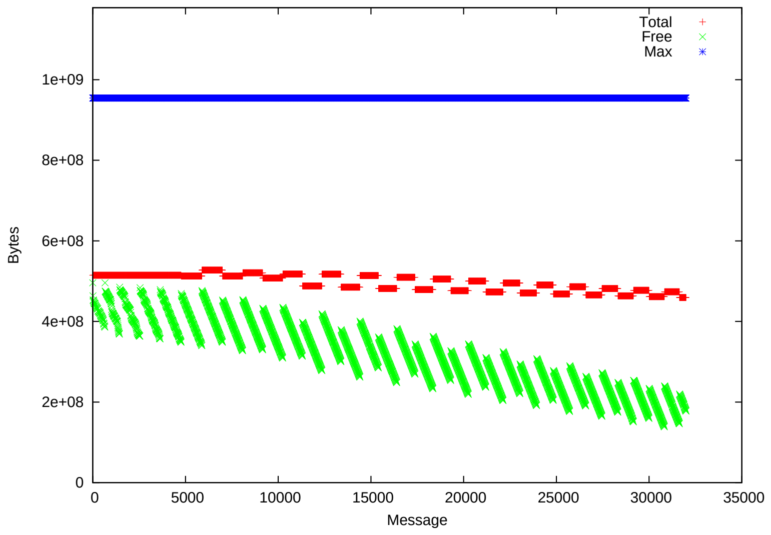

Figure 50: UB_{prio}, 10 msg/s, node2

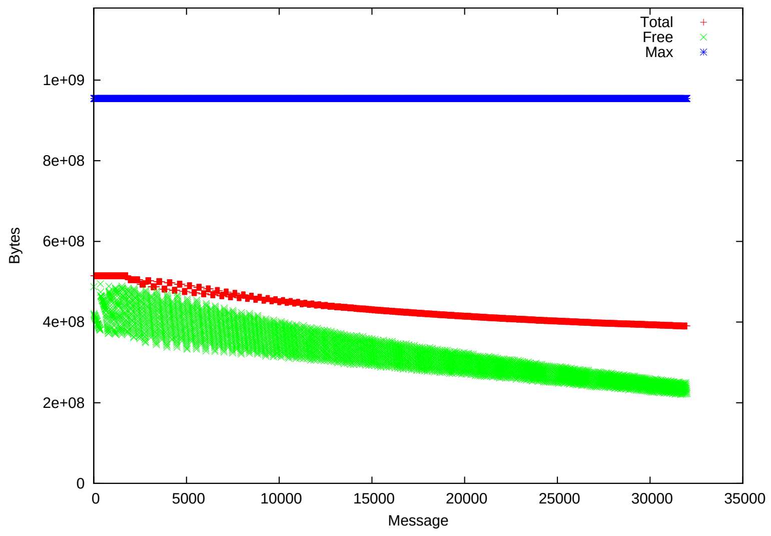

Figure 51: TR, 10 msg/s, node1

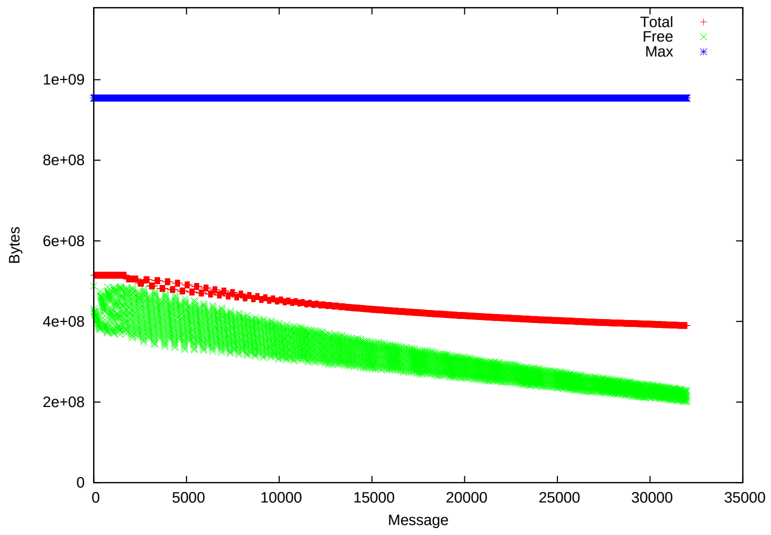

Figure 52: TR_{prio}, 10 msg/s, node1

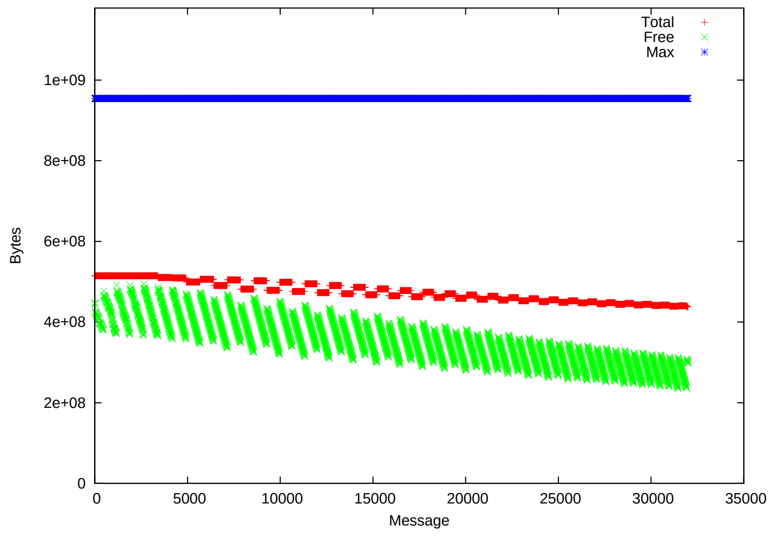

Figure 53: CH, 10 msg/s, node1

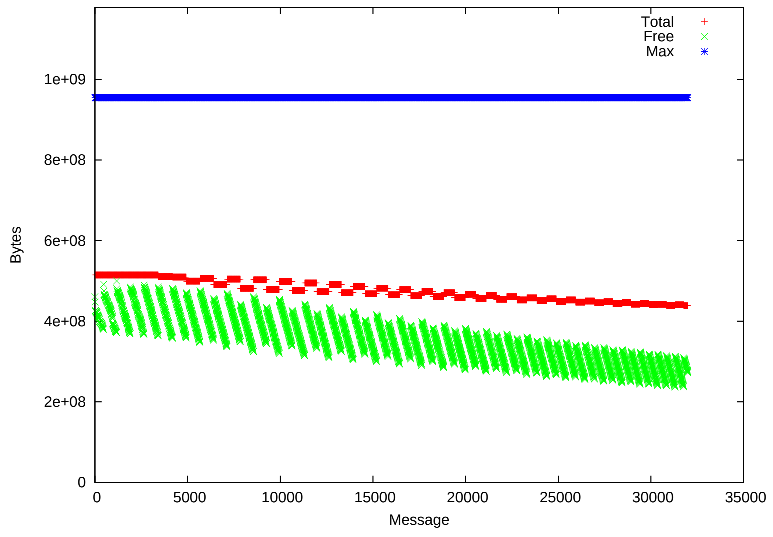

Figure 54: CH_{prio}, 10 msg/s, node1

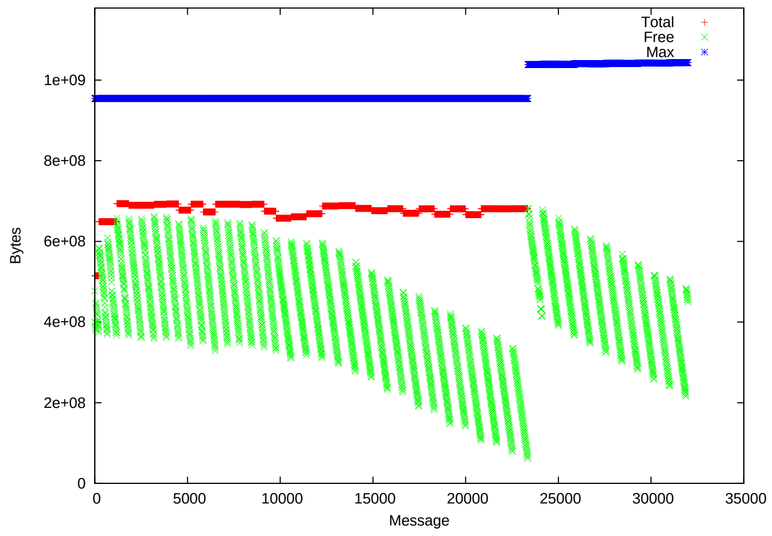

Figure 55: UB, 40 msg/s, node1 (sequencer)

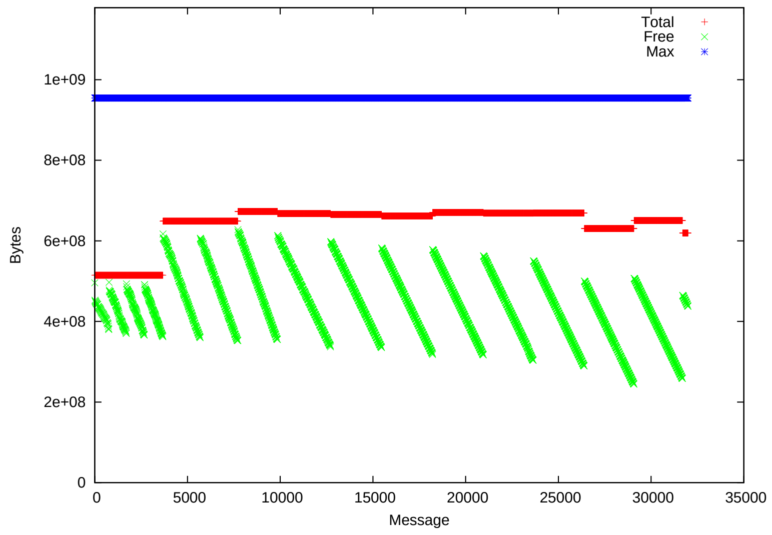

Figure 56: UB, 40 msg/s, node2

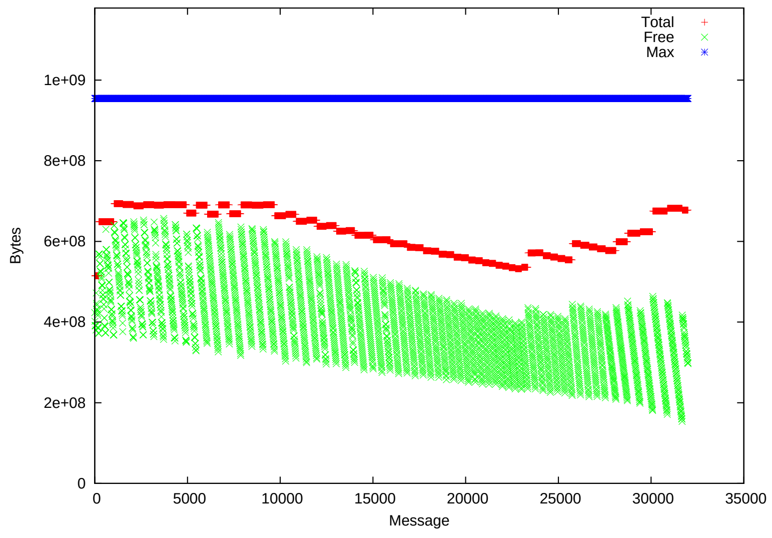

Figure 57: UB_{prio}, 40 msg/s, node1 (sequencer)

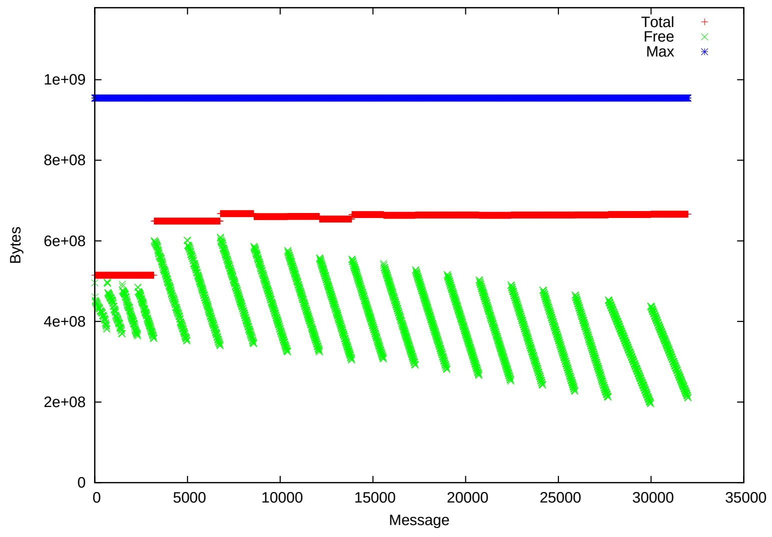

Figure 58: UB_{prio}, 40 msg/s, node2

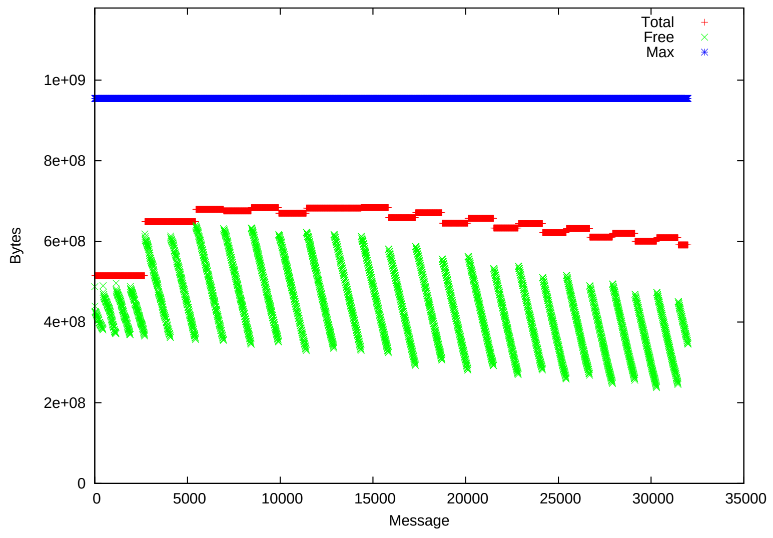

Figure 59: TR, 40 msg/s, node1

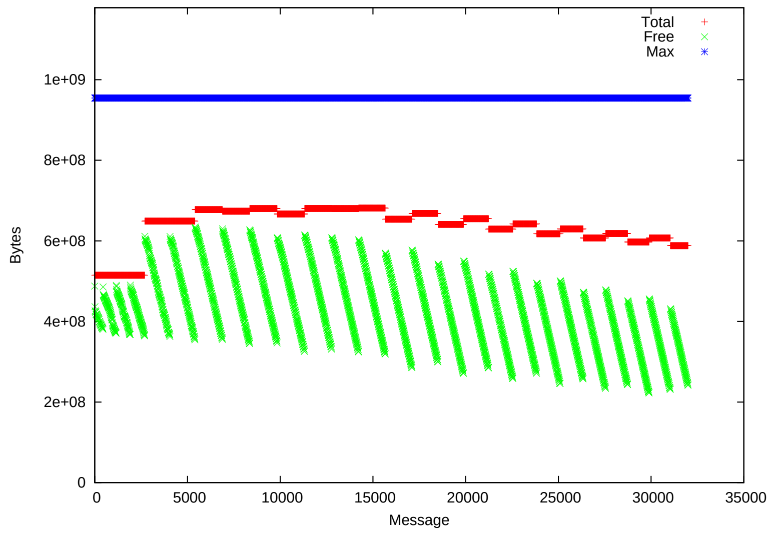

Figure 60: TR_{prio}, 40 msg/s, node1

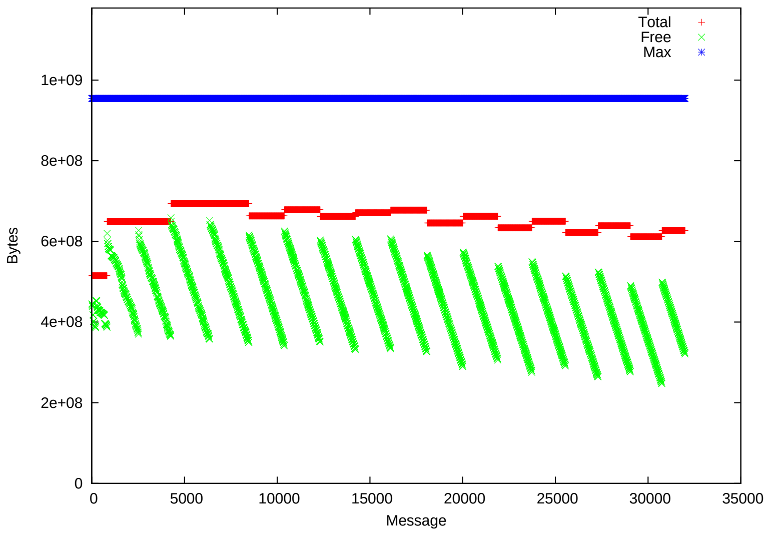

Figure 61: CH, 40 msg/s, node1

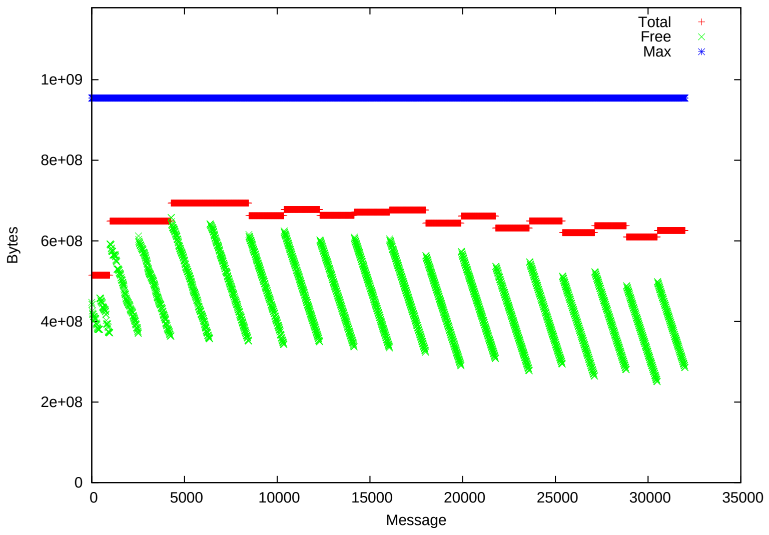

Figure 62: CH_{prio}, 40 msg/s, node1

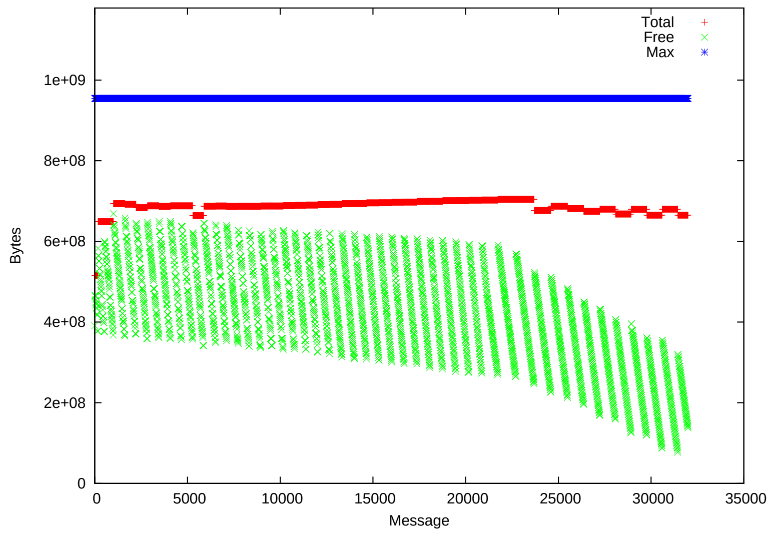

Figure 63: UB, 60 msg/s, node1 (sequencer)

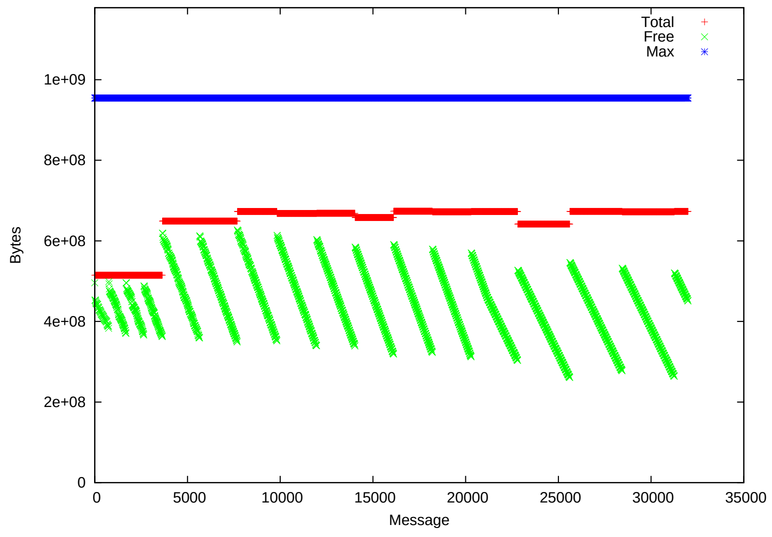

Figure 64: UB, 60 msg/s, node2

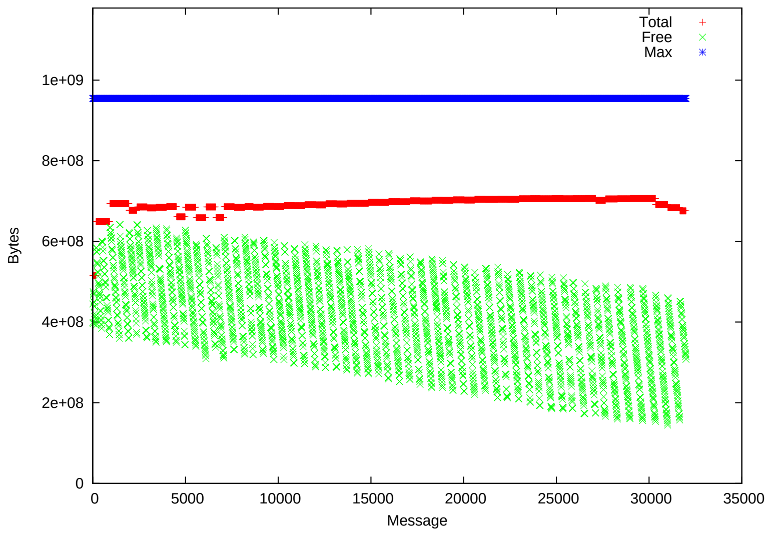

Figure 65: UB_{prio}, 60 msg/s, node1 (sequencer)

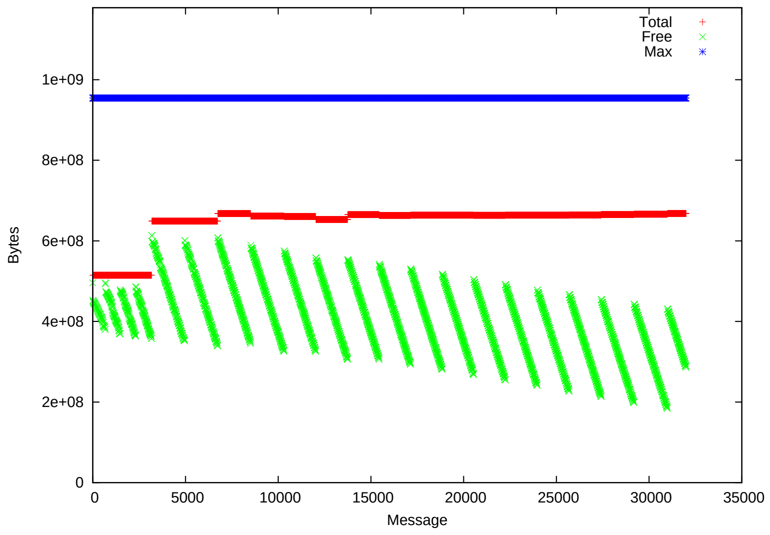

Figure 66: UB_{prio}, 60 msg/s, node2

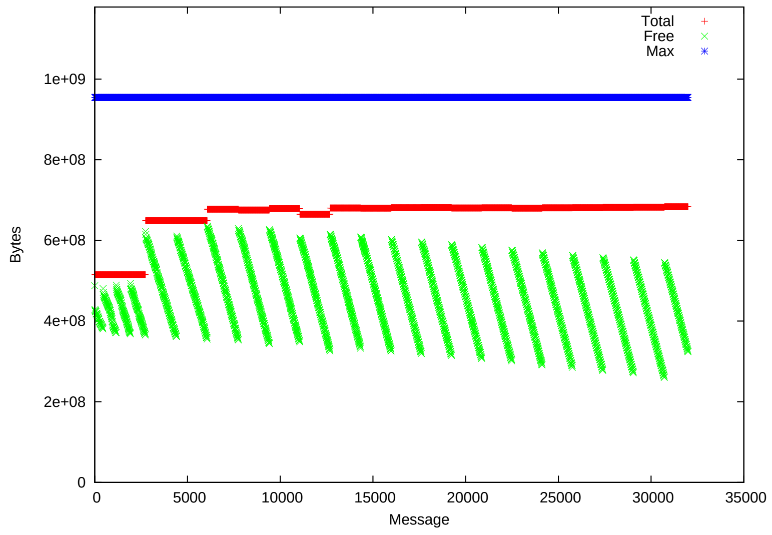

Figure 67: TR, 60 msg/s, node1

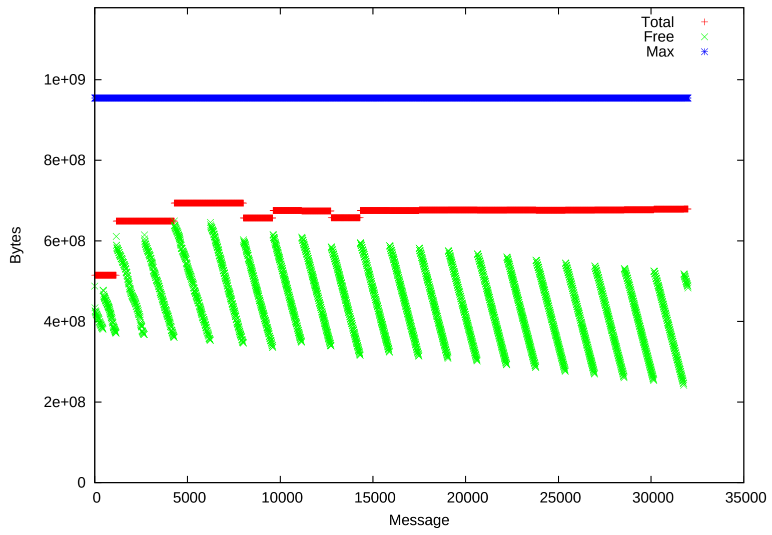

Figure 68: TR_{prio}, 60 msg/s, node1

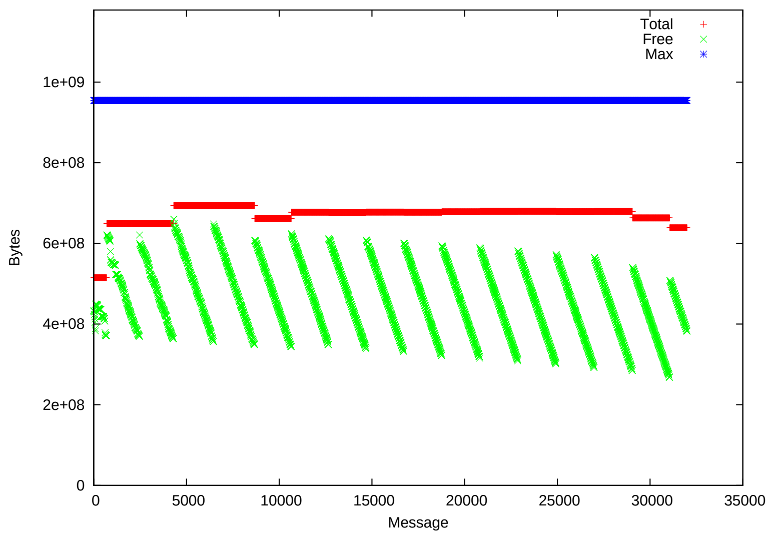

Figure 69: CH, 60 msg/s, node1

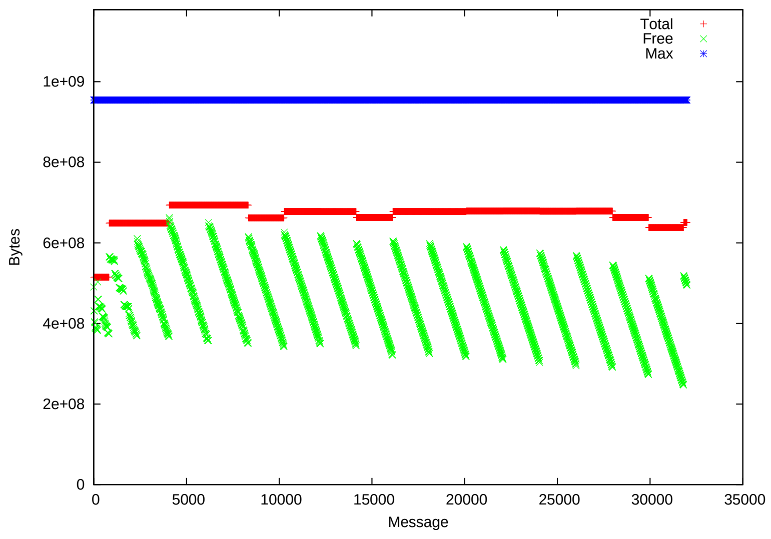

Figure 70: CH_{prio}, 60 msg/s, node1

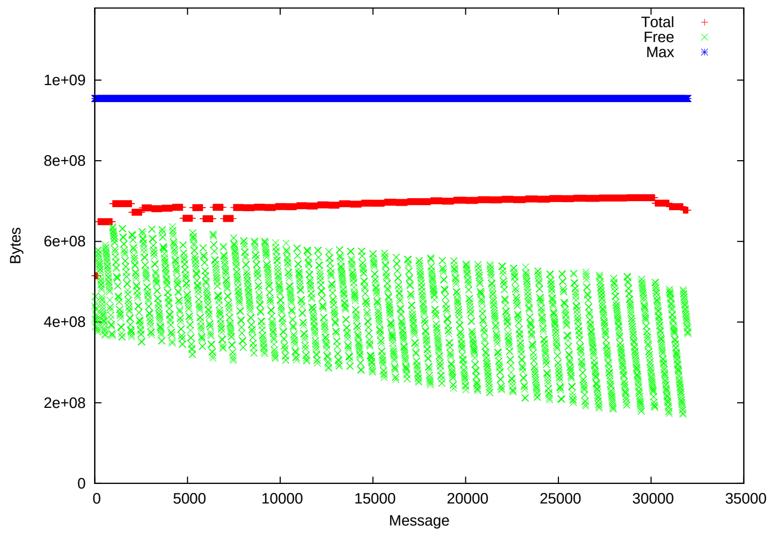

Figure 71: UB, 80 msg/s, node1 (sequencer)

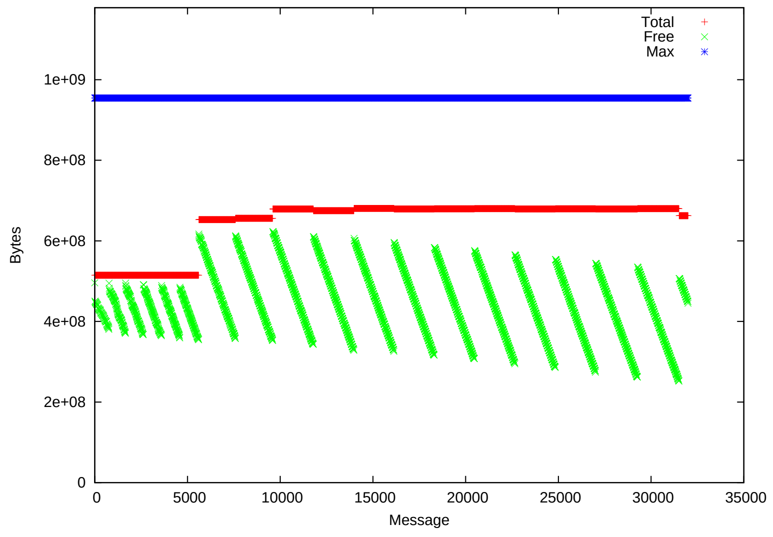

Figure 72: UB, 80 msg/s, node2

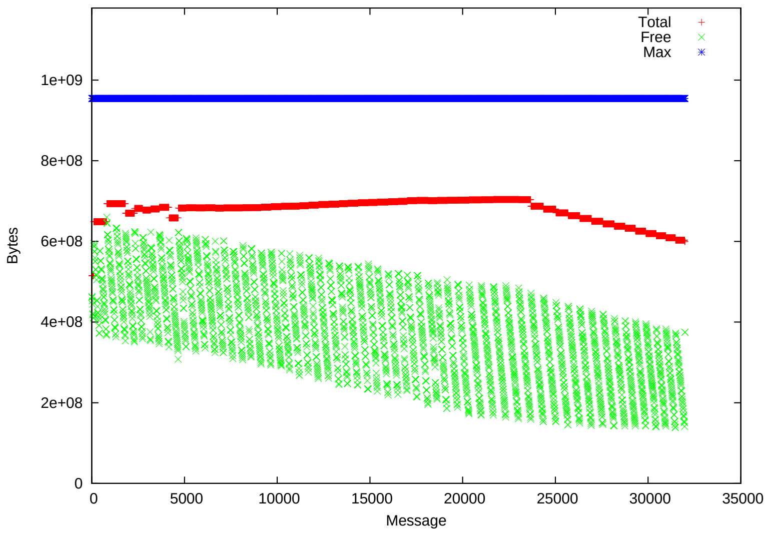

Figure 73: UB_{prio}, 80 msg/s, node1 (sequencer)

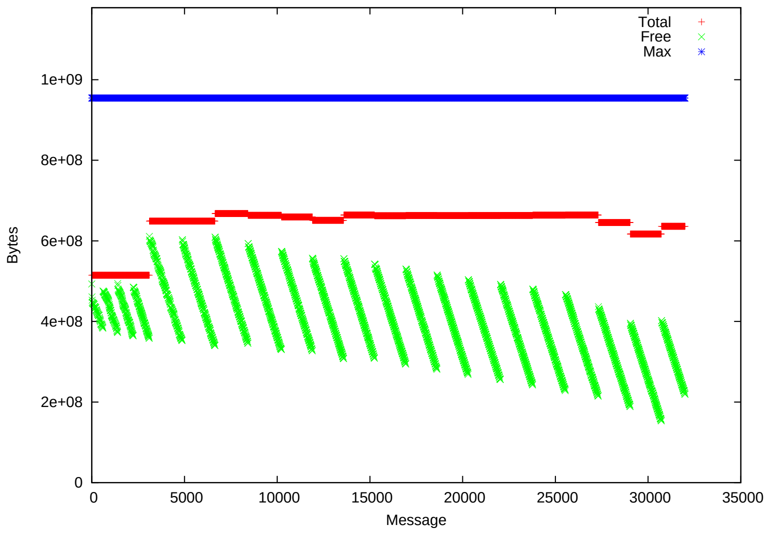

Figure 74: UB_{prio}, 80 msg/s, node2

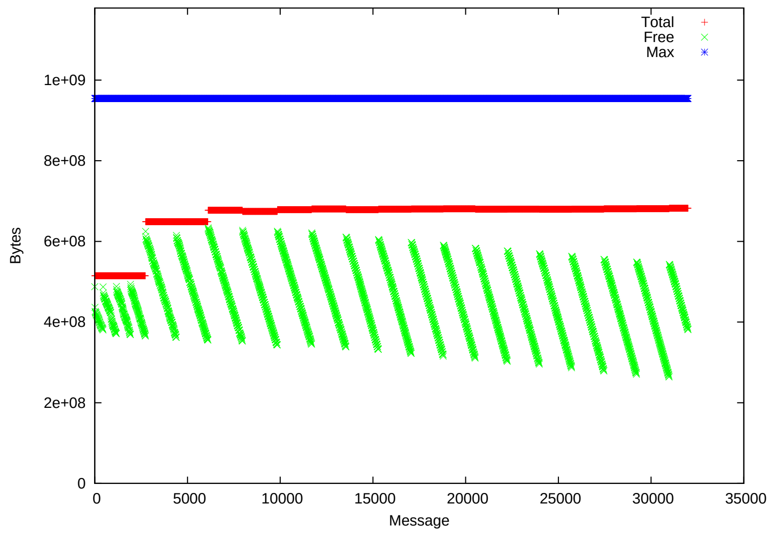

Figure 75: TR, 80 msg/s, node1

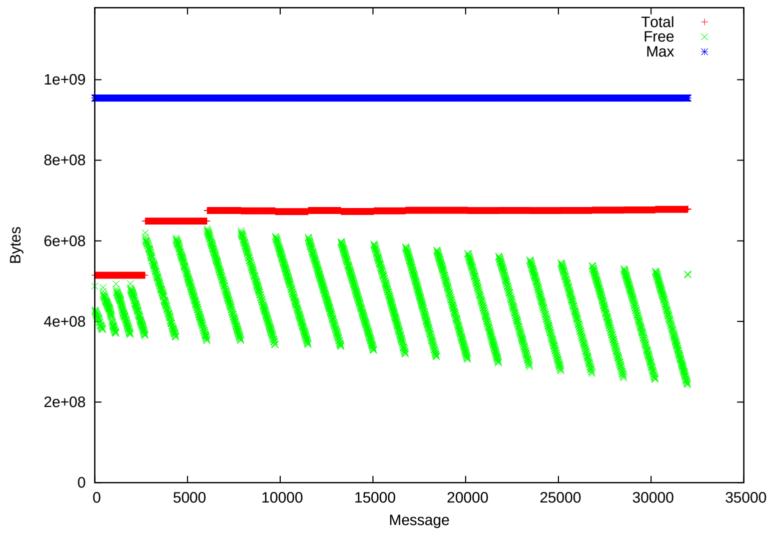

Figure 76: TR_{prio}, 80 msg/s, node1

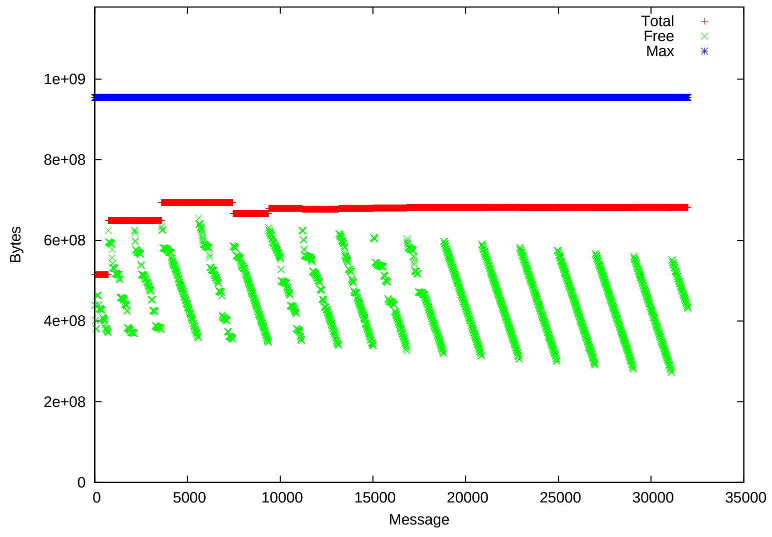

Figure 77: CH, 80 msg/s, node1

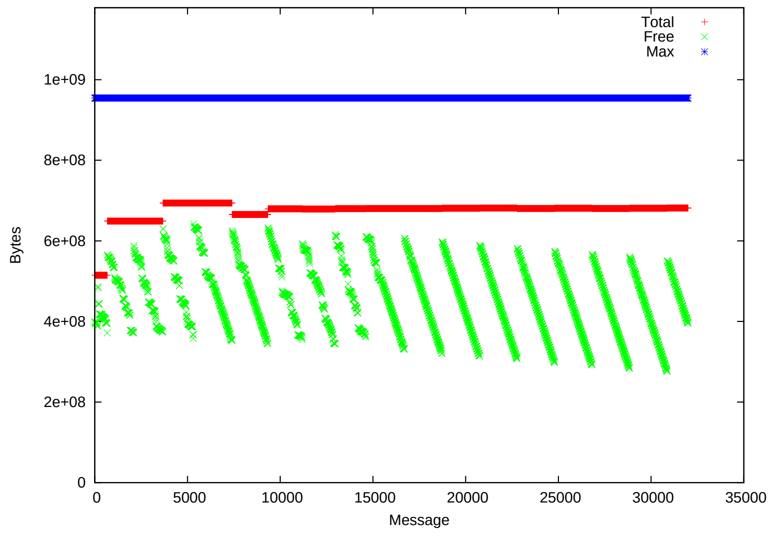

Figure 78: CH_{prio}, 80 msg/s, node1

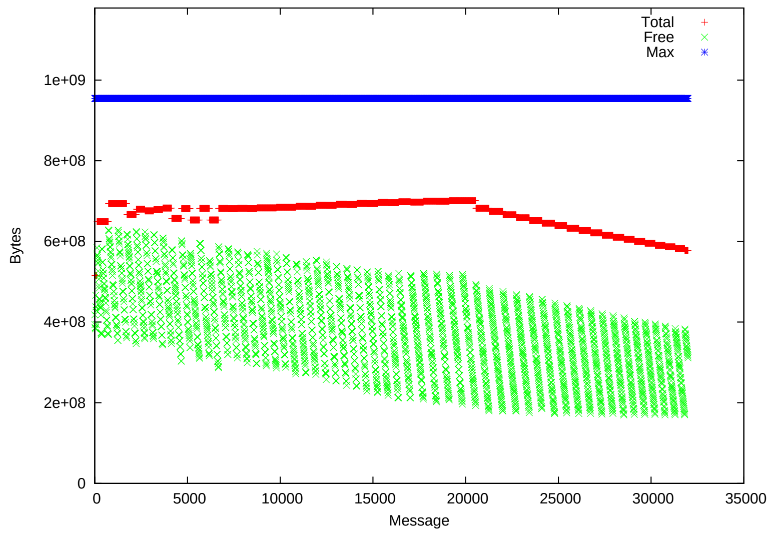

Figure 79: UB, 100 msg/s, node1 (sequencer)

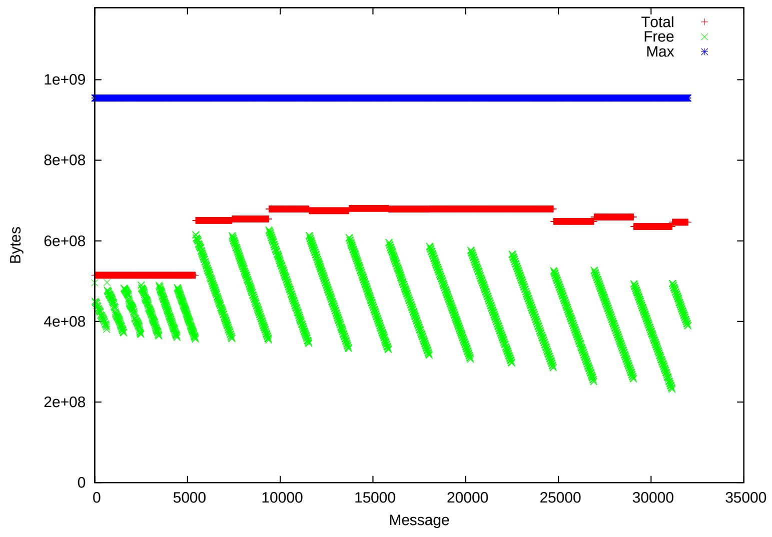

Figure 80: UB, 100 msg/s, node2

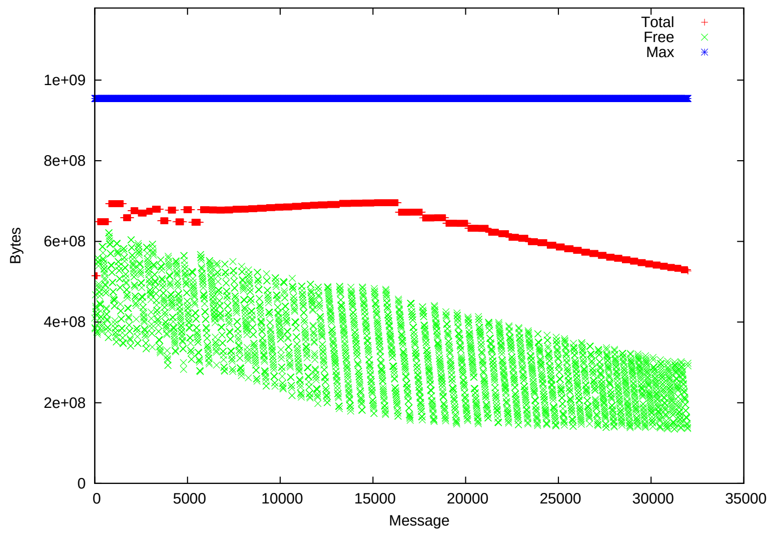

Figure 81: UB_{prio}, 100 msg/s, node1 (sequencer)

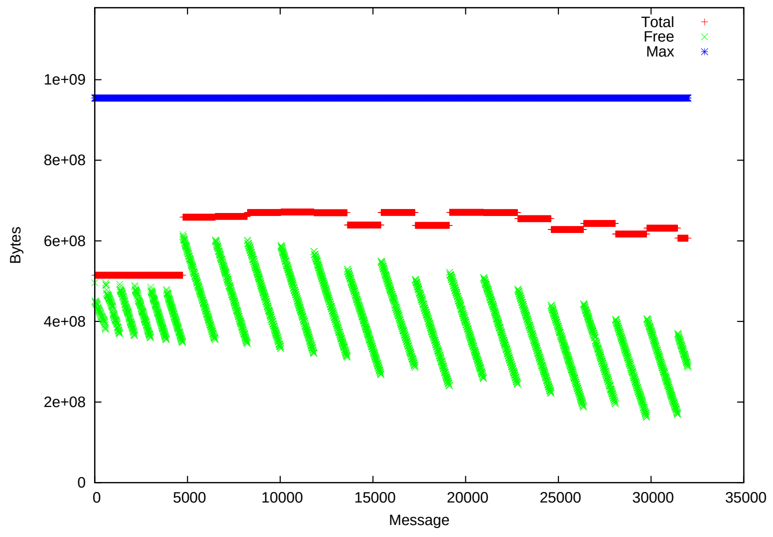

Figure 82: UB_{prio}, 100 msg/s, node2

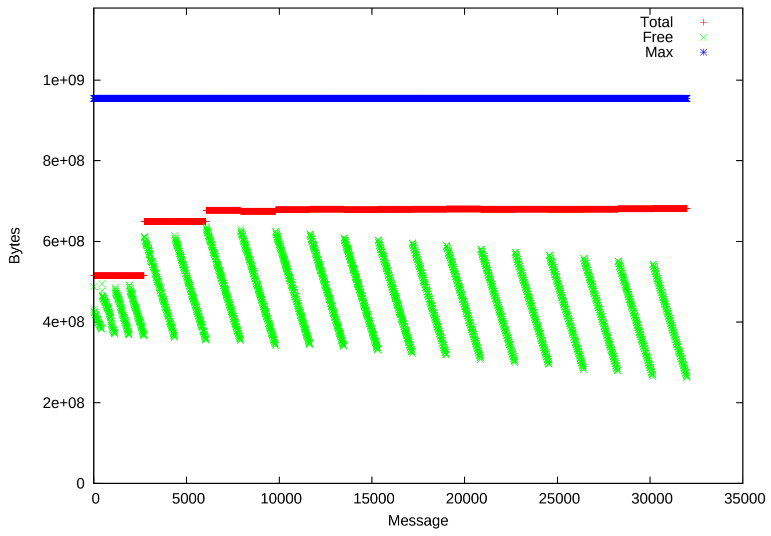

Figure 83: TR, 100 msg/s, node1

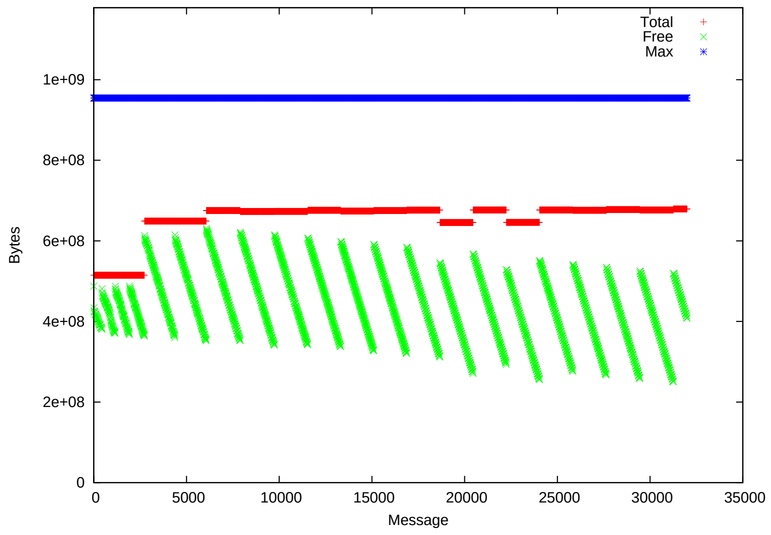

Figure 84: TR_{prio}, 100 msg/s, node1

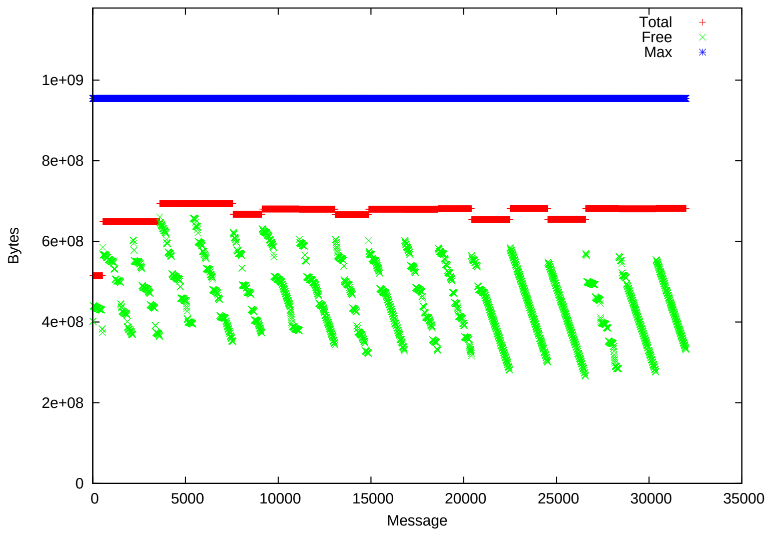

Figure 85: CH, 100 msg/s, node1

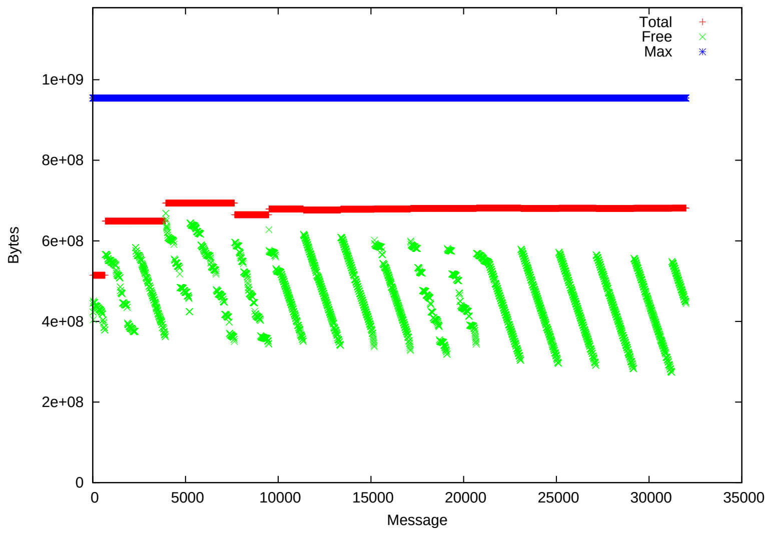

Figure 86: CH_{prio}, 100 msg/s, node1

References

[1] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN*, pages 327–336, 2000.

[2] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[3] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[4] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[5] Xavier Défago, André Schiper, and Péter Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[6] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[7] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[8] Luis Irún-Briz, Rubén de Juan-Marín, Francisco Castro-Company, Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. In *European Conference on Parallel and Distributed Computing (Euro-Par 2005)*, pages 349–359, Lisbon, Portugal, Sept. 2005.

[9] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the amoeba group communication system. In *16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 436–448, Washington, DC, USA, 1996. IEEE Computer Society.

[10] Emili Miedes, Francesc D. Muñoz, and Hendrik Decker. Reducing transaction abort rates with prioritized atomic multicast protocols. In *European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, pages 394–403, Las Palmas de Gran Canaria, Spain, 2008.

[11] Emili Miedes and Francesc D. Muñoz-Escoí. Managing priorities in atomic multicast protocols. In *ARES: Intl. Conf. on Availability, Reliability and Security*, pages 514–519, Barcelona, Spain, 2008.

[12] Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, R.K. Budhia, and C.A. Lingley-P apadopoulos. Totem: a fault-tolerant multicast group communication system. *Comm. of the ACM*, 39(4):54–63, 1996.

[13] Akihito Nakamura and Makoto Takizawa. Priority-based total and semi-total ordering broadcast protocols. In *12th Intl. Conf. on Dist. Comp. Sys. (ICDCS 92)*, pages 178–185, 1992.

[14] Akihito Nakamura and Makoto Takizawa. Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel netw ork. In *2nd Intl. Symp. on High Performance Dist. Comp.*, pages 281–288, 1993.

[15] Luís Rodrigues, Paulo Veríssimo, and Antonio Casimiro. Priority-based totally ordered multicast. In *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control*, 1995.

[16] Alan Tully and Santosh K. Shrivastava. Preventing state divergence in replicated distributed programs. In *9th Symposium on Reliable Distributed Systems*, pages 104–113, 1990.

[17] Yun Wang, Francisco Brasileiro, Emmanuelle Anceaume, Fabíola Greve, and Michel Hurfin. Avoiding priority inversion on the processing of requests by active replicated servers. In *Dependable Systems and Networks*, pages 97–106. IEEE Computer Society, 2001.

[18] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *SRDS*, pages 206–215, 2000.