Adaptive Message Packing for Group Communication Systems

Alberto Bartoli[1], Cosimo Calabrese[1], Milan Prica[1], Etienne Antoniutti Di Muro[1], and Alberto Montresor[2]

[1] Dip. Elettrotecnica, Elettronica ed Informatica, Università di Trieste (Italy).
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna (Italy).

Abstract. Group communication is one of the main paradigms for implementing replication middleware. The high run-time costs of group communication may constitute a major performance bottleneck for modern enterprise applications. In this paper we investigate the applicability of *message packing*, a technique originally proposed by Friedman and Van Renesse in 1997 for improving the performance of group communication, to modern hardware and group communication toolkits. Most importantly, we extend this technique with a policy for varying the packing degree automatically, based on dynamic estimates of the optimal packing degree. The resulting system is *adaptive* in that it allows exploiting message packing efficiently in a dynamic and potentially unknown run-time environment. Several case studies are analyzed.

1 Introduction

Replication of COTS components is now a widespread way for enhancing dependability of services. Group communication is one of the key middleware technologies for supporting replication [7, 12, 14, 6, 2, 8, 3, 9, 10]. Mainly due to its powerful guarantees, however, group communication usually entails a high cost that plays a decisive role in determining the performance of a replicated application. In our test environment, for example, a non-replicated web service running in Tomcat can sustain a throughput of approximately 300 operations per second. On the other hand, in the same environment, a 3-way replicated application that *only* multicasts and delivers 2000-byte messages with Spread [1] (total order with safe delivery) reaches 100% CPU usage, thereby saturating the system, at a throughput very close to the above. Since a replicated web service requires at least one multicast per operation, it is easy to see that group communication may constitute a major bottleneck for the replicated implementation.

In the attempt of shifting such bottleneck up to higher values, we are investigating techniques for improving the efficiency of existing group communication systems. This paper presents our results in this area. Motivation for this work can be found in the ADAPT project (an EU-funded joined effort between several institutions across Europe and Canada, see http://adapt.ls.fi.upm.es/adapt.htm for more details) whose aim is to develop middleware for constructing replicated services based on the J2EE framework.

This work is supported by Microsoft Research (Cambridge, UK) and the EU IST Programme (Project ADAPT IST-2001-37126).

This goal is part of the broader objective of ADAPT, namely the construction of virtual enterprises out of collections of "basic services" hosted at different, potentially remote, enterprises. In this framework, we are using replication based on group communication as a mean for improving availability and dependability of such basic services. The group communication infrastructure used is Spread, augmented with a thin Java layer on top (as explained later).

The starting point of our work is the proposal by Friedman and Van Renesse [4], in which they demonstrated that *message packing* can significantly improve throughput of total-order protocols. This technique simply consists in buffering application messages for a short period of time before actually sending them as a single message, in order to reduce the overhead caused by the ordering protocol. Their experiments are based on 1997 hardware and, in particular, 10 Mbps Ethernet. Few experiments made it immediately clear that message packing can be very effective even with more modern hardware, including 100 Mbps Ethernet, and even with a group communication system based on a client-daemon architecture (unlike the one used in the cited work).

In this work, however, we exploit message packing in a way quite different from that in [4]. First, we buffer messages until the desired *packing degree* (number of buffered messages) has been reached, irrespective of the amount of time that a message has spent in the buffer. This approach enables us to gain deeper insight into the relationship between *throughput*, *CPU usage*, *latency* and packing degree. Of course, a practical implementation will have to introduce an upper bound to the time a message spends in the buffer, otherwise latency could grow excessively and the system could even stop sending messages (the trade-off between throughput and latency is discussed later in the paper). Second, we have defined an *adaptive* policy for changing the packing degree at run-time and automatically. This is a key issue because the packing degree yielding the best performance depends on a number of factors, including characteristics of the message source, message size, processing load associated with each message, hardware and software platform. Not only these factors can be potentially unknown, they can also vary dynamically at run-time. Selecting the packing degree once and for all can hardly be effective. With our policy, the system automatically determines a packing degree close to the value that happens to be optimal in that specific environment (at least for the cases that we have analyzed exhaustively, detailed in the paper). Moreover, the policy has proven to be robust against occasional variations of additional CPU load induced by other applications.

The resulting behavior of the system is as follows: (i) When the source injects a "low" load, message packing remains inactive. (ii) In the case of a "medium-to-high" load, message packing starts to act leading to higher delivered throughput and *decreased CPU usage*. (iii) In the case of a "very high" load, CPU usage reaches 100% anyway but message packing leads to a higher delivered throughput. Our adaptive policy hence helps the system to *automatically* increase the bottleneck point induced by group communication. Although the effectiveness of our proposal will have to be evaluated within a complete replication solution, we believe that these results are encouraging and these features could be very important in the application domain of interest in ADAPT.

The rest of the paper is organized as follows. Section 2 introduces our system and describes the testing methodology used later in the paper. In Section 3, our adaptive

policy for message packing is presented. Section 4 evaluates the effectiveness of the adaptive policy, by describing several tests under various conditions, while Section 5 discusses our results and concludes the paper.

2 The System

2.1 Group Communication System

Our group communication system, called JBora, consists of an harness around Spread. For the purpose of the present discussion, JBora supports an execution model very similar to the Extended Virtual Synchrony model [11] of Spread. The key difference is that JBora supports a notion of primary partition and does not make transitional views visible to applications. From the implementation point of view, JBora consists of a thin Java layer on top of the Java interface to Spread. One JBora multicast maps to exactly one Spread multicast and the size of the two multicasts is the same, except for a 4-byte JBora specific header. This header is inserted without performing an additional memory-to-memory copy beyond those already performed by the Java interface of Spread. When Spread delivers a multicast to JBora, the multicast is immediately deliverable (except when the multicast is delivered in a transitional view, but this condition does not occur in our experiments). One Spread daemon runs on each replica. A JBora application running on a given node connects to the Spread daemon on that node.

Message packing has been implemented by slightly modifying the portion of JBora that implements the operation `mCast(m)`. Rather than invoking the multicast operation of Spread immediately, `mCast(m)` inserts `m` in a *packing buffer*; then the operation may return either immediately or after multicasting the entire packing buffer as a single Spread multicast. The portion of JBora that implements the `Receive()` operation has been modified to unpack received messages as appropriate. Transmission of the packing buffer occurs when the number of messages in it equals the current value for the *packing degree*, denoted as $pack$. This value can be either defined at configuration time and kept constant across the entire execution, or it can be adjusted dynamicaly based on observed execution statistics (see Section 3). Of course, a practical implementation of message packing has to include further conditions for triggering transmission of the packing buffer. For example, if the number of buffered messages was smaller than $pack$ and then the source stopped generating new messages, then the packing buffer would be never transmitted. This topic will be discussed in Section 5.

2.2 Operating Environment

The operating environment consists of a network of Dell Optiplex GX300 (PIII 800MHz, 512 MB RAM), connected to a 100 Mbps switched Ethernet and running Sun Microsystems' JDK 1.4.0 over Windows 2000 Professional. Each node, hereinafter *replica*, is equipped with JBora.

Each replica runs an application written in Java that maintains a simple replicated logging system. The application consists of two threads: the *source* thread generates messages to be multicast through JBora, while the *receiver* thread receives messages

from JBora and writes them into a MySQL database local to the replica. Writes on the database are all done into the same database table. This architecture has been adopted to emulate a simplified, yet realistic replicated three-tier application, where one member multicasts the requests received from clients to all replicas, which execute them by interacting with the database tier. Of course, the performance figures that have been obtained depends on the combination of the various pieces of software present in the system, but we have verified in all experiments below that the bottleneck is indeed the group communication system, not the database.

Each experiment below refers to a system composed of three replicas where only one of them generates messages. We focussed on a small number of replicas because the use of group communication that we are pursuing in ADAPT is for improving fault-tolerance and we believe that, in practical environments, only small replication degrees are likely to be used. We focussed on a single replica that generates messages only for restricting the number of parameters to investigate.

Our experiments are based on sources quite different from those in [4] and much closer to our needs. First, we used total order with *safe* delivery (also called *uniform* delivery). These are the strongest delivery guarantees normally offered by group communication platforms, and also those that are most demanding at run-time. We intend to design replication algorithms based on safe delivery because, without this guarantee, coping with certain failure patterns correctly would require complex and costly actions (e.g., when the sender of a multicast is the only replica that receives that multicast [5]). Second, we considered sources that generate messages continuously or at bursts (alternating a burst with a sleeping time). That is, unlike [4], we do not constrain the generation of new messages by the delivery of messages multicast by other replicas. This is because in our intended application domain generation of new multicasts is triggered by the arrival of operation requests from remote clients, i.e., an event that can occur potentially at any time and usually does not depend on the arrival of multicasts from other replicas. The resulting scenario simulates a situation in which on the sending replica there are always new messages waiting to be multicast. We have implemented a flow control mechanism that suspends the source thread when the load injected into the group communication system is excessive (without this mechanism, the sending replica takes an exception and is forcibly expelled from the group).

2.3 Measurements

For each experiment we have measured *throughput*, *latency* and *CPU usage*. Throughput has been measured as L/N, where L is the time interval between the receiving of the last message and the receiving of the first message. This time interval has been measured at the receiving thread of the sending replica. Quantity N is the number of messages in the experiment run, approximately 25000 in each case. We have not used the standard timer available in Java through the `System` class, because its resolution (approximately 16 msec) was not sufficient for our measurements, in particular for those of latency. Instead, we have used a publicly available timing library that exploits system-specific hooks and allows measuring time intervals with a resolution of 1 microsec [13].

Message size (bytes)	*Throughput (msg/sec)*	*Latency (msec)*
100	596	11.07
1000	453	7.08
10000	114	47.28

Table 1. Performance without message packing.

Message size (bytes)	*Optimal*	*Throughput (msg/sec)*	*Improvement*	*Latency (msec)*
100	28	5666	9.51	55.24
1000	11	1577	3.48	28.16
10000	3	201	1.76	29.37

Table 2. Maximum performance (throughput) obtained with message packing.

The latency for an experiment run is the average latency amongst all messages of that run. The latency of each message has been measured at the sending replica, as follows. The sender thread of the sending replica reads the timer immediately before invoking the Spread multicast operation and inserts the corresponding value in the message to be multicast. The receiver thread of the sending replica reads the timer as soon as it has received a message. The difference between this value and the one contained in the message is the latency value for that message. Note, the time spent in the packing buffer is taken into account for evaluating the latency of each individual message. Average CPU usage has been estimated by visually inspecting the task manager of the Windows 2000 operating system.

3 An Adaptive Policy for Message Packing

Our first suite of experiments used a source thread that *continuously* generates fixed-size messages, putting JBora under stress. The results obtained with message packing disabled are shown in Table 1, for three different message sizes (100,1000,10000). These results constitute the baseline for comparing the results obtained through message packing. Then we made a number of experiments with message packing enabled. In each experiment we kept $pack$ constant. The results for 1000-byte messages are in Figure 1. It can be seen that throughput increases substantially, reaching a maximum of 1577 msg/sec with $pack = 11$. This represents an improvement of 3,48 times over the throughput without packing. For sake of brevity, we omit the figures for 100-byte and 10000-byte messages: the curves have the same shape as Figure 1 with numerical values that depend on the message size. A summary is given in Table 2. In all cases throughput increases substantially, at the expense of latency (see also Section 5).

Although message packing may be very effective in improving throughput, a key problem is determining the suitable value for the packing degree $pack$. Our experiments clearly show that the optimum value greatly depends on the message size. Moreover, a realistic source will generate messages of varying sizes. Finally, and most importantly, the effect of message packing may greatly depend on a number of factors that can

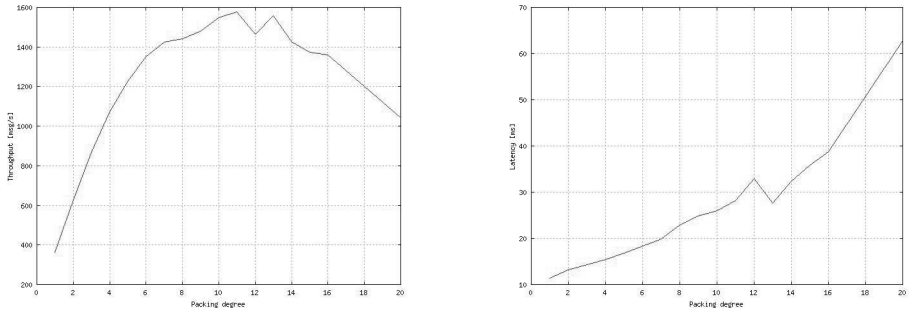

Fig. 1. Throughput (left) and latency (right) with varying packing degrees (msg size is 1000 bytes).

vary dynamically and usually cannot be predicted in advance, for example, the load on replicas induced by other activities and the specific hardware/software environment. Determining one single value for $pack$ once and for all can hardly be effective.

We have implemented a simple and inexpensive *mechanism* for varying $pack$ dynamically. Each replica measures the throughput with respect to multicasts generated by that replica at regular intervals, every T_a seconds (the throughput is averaged over this interval). Based on these observed statistics, each replica may vary $pack$ dynamically, trying to adapt to the features of the operating environment. In all of the experiments reported here we set $T_a = 5$sec and we updated the packing degree based on the last two measures, i.e., every 10 sec. The problem, of course, is determining an effective *policy* for exploiting this mechanism.

We experimented with policies that implement the following basic rules ($throughput_i$ denotes the i-th throughput measurement, $pack_i$ denotes the i-th packing degree):

1. Initially, $pack = 1$ (i.e., no packing enabled);
2. $pack$ may only be incremented by 1 or decremented by 1;
3. $pack \in [1, packmax]$;
4. $throughput_i = throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i$ (steady state);

The issue is defining the *update rule* that varies $pack$ so as to improve throughput. From the shape of the curves throughput vs. $pack$, it would appear that defining such rule is simple: one could simply increase $pack$ when throughput increases and decrease $pack$ otherwise:

- $throughput_i > throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i + 1$
- $throughput_i < throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i - 1$

The resulting simple policy indeed converges quickly to the value for $pack$ that is optimal for the specific message size that characterizes the source. By optimal value we mean the one that we have determined previously, by exhaustive testing (e.g., for 1000-byte messages the optimum value is 11). This is a valuable result because, of course, the system does not know that value but finds it automatically. The quickness in reaching this value depends on how frequently the throughput measurements are taken.

Unfortunately, this policy is not sufficiently robust against occasional throughput variations, induced for example by short additional loads. In many executions, $pack$ falls back to its minimum value 1. The reason is as follows. Suppose the optimal value has not been reached yet and $throughput_i$ is lower than $throughput_{i-1}$ because of a transient phenomenon out of control of the group communication system. In this case, the packing degree would be lowered. At this point it is very likely that the next measurement will show an even lower throughput, thereby ending up quickly with $pack = 1$.

It is also possible that throughput collapses because $pack$ oscillates around excessively high values. To realize this, consider Figure 2. Suppose the system be characterized by curve B and the packing degree has reached its optimal value pB. Next suppose that, due to some additional load, the system be characterized by curve A. The next measure will show that throughput has decreased, thus $pack$ will be decremented to $pB - 1$. The next measure will then show that throughput has increased, thus $pack$ will be incremented again at pB. Since this increment will cause throughput to decrease, at this point the value of $pack$ will keep on oscillating around pB, a value that may be largely suboptimal in curve A. Note, phenomena similar to those just discussed could occur even if the source changed the message size during the run.

For these reasons, we experimented with a simple refinement of the above update rule. The basic idea is this: one has to make sure that when $pack$ starts to decrease, it may continue decreasing only if throughput grows — i.e., only when $pack$ is indeed greater than the optimal value corresponding to the peak throughput. Otherwise, $pack$ should no longer decrease and should increase instead. We implement this idea with the following update rule:

- $pack_i \geq pack_{i-1} \Longrightarrow$ *// Increase $pack$ when $throughput$ increases*
  - $throughput_i > throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i + 1;$
  - $throughput_i < throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i - 1;$
- $pack_i < pack_{i-1} \Longrightarrow$ *// Decrease $pack$ when $throughput$ increases*
  - $throughput_i > throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i - 1;$
  - $throughput_i < throughput_{i-1} \Longrightarrow pack_{i+1} := pack_i + 1;$

It is simple to realize that, as confirmed by our experiments, this policy prevents the instability behaviors described above. Short transient loads may provoke a decrease of the packing degree, but not its collapsing to the minimum value. Once the additional load has disappeared, the packing degree converges again to its previous value. Similarly, the packing degree does not oscillate around excessively high values. The policy is thus quite robust. All the results presented later are based on this policy.

4 Evaluation of the Adaptive Policy

4.1 Continuous Source

The most demanding test for a group communication system is given by a source thread that *continuously* generates new multicasts. We have evaluated this scenario with both

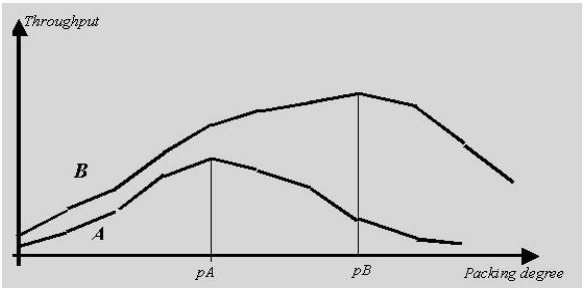

Fig. 2. Throughput vs. Packing degree for different loads: curve A corresponds to an higher load than curve B (curves for differing message sizes and/or additional load have the same shape).

messages of fixed size, and with messages of variable size. All the results in this section corresponds to a CPU usage close to 100%. That is, nearly all of the CPU time on the sending replica is spent for propagating messages (recall that each message is logged on a MySQL database, though).

Messages with Fixed Size With the policy enabled and 1000-byte messages, the packing degree oscillates between 10 and 12 and the average throughput is 1338 msg/sec. This corresponds to 85% of the throughput obtained with the packing degree statically set to its optimal value 11. It also corresponds to almost a 300% throughput improvement over the system without packing enabled. Latency is 19.58 ms.

The time it takes to $pack$ for reaching the 10-12 range from its initial value 1 is approximately 100 seconds. The reason is, we update the packing degree every 10 seconds and we can only change it by 1 at every step. We did not experiment with more aggressive policies attempting to shorten this interval. This could be done with shorter update intervals, e.g., 2-3 seconds. We believe that altering the packing degree by more than one unit could make the policy less stable with more realistic sources and environments. We leave this topic open for further investigation and will not mention this issue any further here.

The reason why the packing degree $pack$ does not remain constant but oscillates around the optimal value is because consecutive throughput measures, in practice, will never show exactly the same result. We could filter this effect out by updating $pack$ only when the difference between consecutive measures falls outside some threshold. Although this approach could increase the average throughput further, it would also introduce another parameter to define and to possibly tune. We preferred to avoid this in the attempt to make a system that requires no magic constants and can tune itself automatically, albeit in a slightly sub-optimal way.

Messages with Variable Size We have performed experiments with a source that injects messages continuously, but with differing sizes. Below we present results for the case in which the source generates 300.000 1000-byte messages followed by 300.000 3000-byte messages and then repeats this pattern indefinitely.

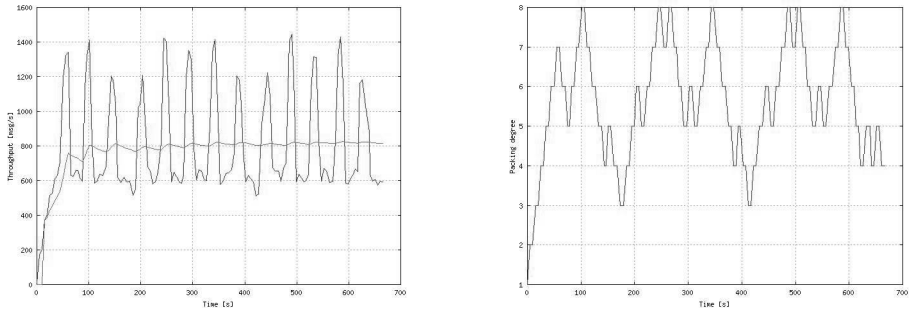

Fig. 3. Average throughput and "instantaneous" throughput over time (left). Packing degree over time (right).

First we have performed a set of experiments for measuring the throughput as a function of $pack$, by keeping the packing degree constant in each run. We have found that throughput without packing is 269 msg/sec, that the maximum throughput is obtained with $pack = 6$ and corresponds to 901 msg/sec. Then we have exercised the system with our policy enabled. Figure 3-left shows the throughput measurements. The flat line shows the average throughput, averaged since the beginning of the experiment. The other line shows the "instantaneous" throughput, i.e., averaged over the last 5 seconds. It can be seen that the average throughput reaches a value close to 800 msg/sec in less than 1 minute and then remains stable despite the variations of the source at around 810 msg/sec. This value corresponds to approximately 90% of the maximum throughput, obtained with $pack$ immutable and fixed a priori to 6. It also corresponds to an almost 300% throughput improvement over the system without packing. Figure 3-right shows the variations of $pack$ over a short time interval.

4.2 Bursty Sources

We have performed experiments with a *bursty* source. The source thread generates a burst of 15 1000-byte messages, sleeps for 20 msecs and then repeats this pattern indefinitely. These experiments are important not only because the source is less extreme than the continuous source discussed above, but also because in this case the CPU usage is smaller than 100%. This scenario should be closer to practical applications where substantial resources are required beyond those consumed by group communication, for example, replication of J2EE components.

These experiments shows an important finding: CPU usage varies with the packing degree $pack$ in a way that is roughly opposite to throughput. That is, the packing degree resulting in peak throughput also results in minimum CPU usage. It follows that message packing may greatly help in improving the overall performance of a complete application, because it contributes to decrease the CPU time required by the replication infrastructure. Another important finding is that our policy for adapting the packing degree automatically works also in this case and indeed decreases significantly the CPU usage.

First we have performed a set of experiment runs by keeping the packing degree constant in each run. The results are in Figure 4. Without packing enabled, CPU usage is 85% and throughput is approximately 260 msgs/sec. With packing enabled, the maximum average throughput is obtained with $pack = 8$ and corresponds to 660 msg/sec. Note that in this situation CPU usage has dropped to 65%.

Then we have run the system with our automatic policy enabled. The average throughput reaches a value close to 430 msg/sec in slightly more than 1 minute. The CPU usage remains below 60%. The packing degree remains stable around two values: 6 for some time intervals and 10 for some others. This behavior is probably due to the fact that the curve throughput vs. packing (Figure 4-right) does not exhibit a single peak and is more irregular than the curves analyzed in the previous section. In summary, the policy increases the throughput by 165% and lets the CPU usage drop from 85% to less than 60%.

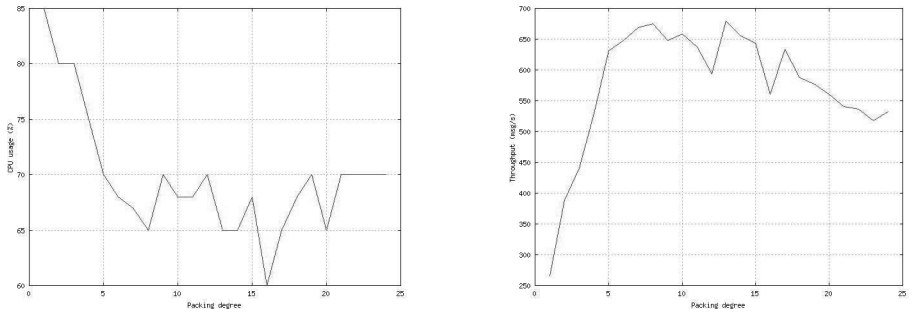

Fig. 4. Average CPU usage (left) and average throughput (right) over packing degree. Bursty source: 15 messages and then 20 msecs sleeping time.

4.3 Short Bursts

Finally, we have investigated the behavior of the system with very short bursts of 1000-byte messages. We made a number of experiments varying the number of messages in each burst and the sleeping time between bursts. Roughly, we have found that as long as the rate of generation of new messages is above 250 msgs/sec, our automatic policy still increases throughput and decreases CPU usage. Below such rate our policy has no effect. Two of the combinations burst length/sleeping time where the policy has effect are given in Table 3.

It is not surprising that when the throughput injected into the system is sufficiently low, message packing has effect neither on throughput nor on CPU usage — it can only increase latency. However, the overall result is significant: when the injected load is sufficiently low, the system is capable of sustaining such load autonomously; when the injected load is not so low, our adaptive policy automatically helps the system in sustaining that load, by shifting the group communication bottleneck to higher loads.

Source	*Scenario*	*Throughput (msg/sec)*	*Latency (msec))*	*CPU usage*
5 msgs every 20 msec	No packing	154	10	60%
	Policy enabled	236	18	45%
2 msgs every 5 msec	No packing	285	5.9	65%
	Policy enabled	319	11	45%

Table 3. Results with bursty source, very short bursts.

Indeed, these experiments allowed us to identify an issue where our policy needs some refinement. The curve throughput vs. packing degree shows a step when $pack$ becomes greater than 1 and then remains more or less flat for a wide range of values of $pack$. It follows that, with the current policy, the packing degree exhibits fairly wide oscillations. Although the resulting behavior is still satisfactory, it seems that a smarter policy is required.

5 Concluding remarks

Friedman and Van Renesse demonstrated in 1997 that message packing can be very effective in improving throughput of group communication systems. Our experiments show that this argument still holds with more modern hardware (including 100 Mbps Ethernet) and when safe delivery is required. Most importantly, we have shown that one can exploit message packing *adaptively*, by means of a simple policy that dynamically matches the packing degree to the specific and potentially unknown characteristics of the message source. Our proposed policy is based on a simple and inexpensive mechanism and has proven to be robust against dynamic and unpredictable changes in the run-time environment.

Of course, message packing is most effective when the source is demanding. In this respect, the best results are obtained when the source injects a very high load for a very long time. However, we have seen that message packing is effective even with sources that inject relatively short message bursts.

We have investigated the effects of message packing even in scenarios when the CPU usage is well below 100%, to simulate a situation in which the group communication system is part of a complex and demanding application based on replication, e.g., replication of J2EE components. We have observed that even in this case message packing can improve throughput substantially and, most importantly, while *decreasing* CPU usage. The main drawback of message packing is that it tends to increase latency, presenting an important trade-off between this quantity and throughput. While our proposed mechanism and policy for message packing are certainly to be evaluated in the context of a complete replication solution, we believe they constitute indeed a promising approach.

To put this claim in perspective, we report data collected from a prototype of a replicated web service that we have developed on top of JBora. The service implements a counter private of each client. Updates to the counter are multicast to each replica. The service implementation does *not* support message packing yet. Clients access the

service through SOAP over HTTP. We simulated a varying number of clients with a publicly-available tool (`http://grinder.sourceforge.net/`). Each client executes an endless loop in which it sends a request and parses the matching response. The data below refer to the same environment used previously, 3 replicas, clients and replicas are on the same Ethernet and all operations are updates. In the range 80-120 clients, throughput grows linearly from 80 to 225 operations/s whereas latency decreases from 900 ms to 416 ms. In the range up 200 clients, throughput remains in the range 200-250 operations/s whereas latency remains in the range 400-550 ms.

We instrumented this replicated web service so as to record the time instants at which each multicast operation is invoked. Then we analyzed off-line these data to obtain a rough indication of whether message packing can realistically be exploited or not. Figure 5-left shows the average time it would take to collect a number of multicasts equal to the (hypothetical) packing degree, as a function of the packing degree. For example, with 100 clients one could collect 8 messages in 83 msec, on the average. By comparing these data with the previous results and having observed that in this case the message size is 810 byte (slightly smaller than 1000 bytes, the size used in the previous experiments), we note that insisting on achieving the optimal packing degree could not be realistic, because the time spent in the packing buffer could grow excessively — well beyond 100 ms. On the other hand, it also appears that message packing can indeed be applied significantly.

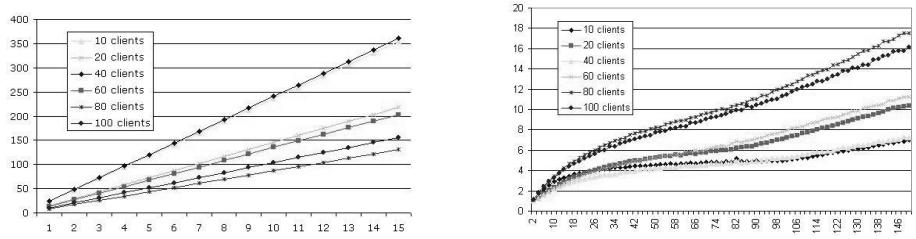

Fig. 5. Average time for filling a specified packing degree (left) and average number of multicasts as a function of the packing interval (right). Times are in ms.

JBora is being extended in order to trigger transmission when either the current packing degree $pack$ has been reached or one of the messages has been in the packing buffer for a time larger than a predefined *packing interval* $pack_{TIME}$. The duration of the packing interval is defined statically, depending on the latency requirements of the application (see also below). This additional condition is capable to handle very irregular sources without introducing unacceptably high delays within the packing buffer (as well as sources that could not even fill the packing buffer). To gain insights into this issue, we analyzed the above data as if the packing interval was the only condition that triggers the transmission. The results are in Figure 5-right, that shows the average number of multicasts in each (hypothetical) packing interval. The average has been

done with respect to the packing intervals that contain at least one multicast. For example, with $pack_{TIME}$ set to 70 ms and 100 clients, each non-empty packing interval would contain 9 multicasts, on the average. Figure 6-right shows the maximum number of multicasts rather than the average, whereas Figure 6-left shows the percentage of non-empty packing intervals. These data indeed confirm that message packing can be realistically exploited, albeit probably in a sub-optimal way.

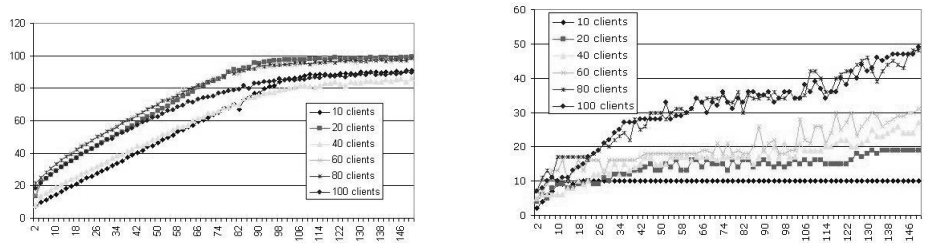

Fig. 6. Percentage of non-empty packing intervals (left) and maximum number of multicasts (right) as a function of the packing interval. Times are in ms.

As future work, we intend to complete the implementation of message packing in order to fully evaluate its effectiveness in the above replicated web service. We are also going to investigate whether allowing the length of the packing interval to vary dynamically and automatically between two or three predefined values is both simple and desirbale. A further issue that deserves more investigation is whether a more aggressive policy that follows more quickly dynamic variations in the run-time environment is really required and can be implemented without detailed, a-priori knowledge about the environment itself.

References

1. Y. Amir and J. Stanton. The spread wide-area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, 1998. `http://www.spread.org`.
2. Y. Amir and C. Tutu. From total order to database replication. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, Vienna, Austria, 2002.
3. R. Friedman and E. Hadad. A group object adaptor-based approach to CORBA fault-tolerance. *IEEE Distributed Systems Online*, 2(7), November 2001.
4. R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proc. of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.
5. C. Karamanolis and J. Magee. Client-access protocols for replicated services. *IEEE Transactions on Software Engineering*, 25(1), January/February 1999.
6. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333 – 379, 2000.

7. P. M. Melliar-Smith L. E. Moser and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.
8. S. Labourey and B. Burke. JBoss clustering. Technical report, The JBoss Groups, 2002.
9. S. Mishra, L. Fei, X. Lin, and G. Xing. On group communication support in CORBA. *IEEE Transactions on Parallel and Distributed Systems*, 12(2), February 2001.
10. G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems*, 1999.
11. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.
12. Y. Ren, D. Bakken, T. Courtney, M. Cukier, D. Karr, P. Ruble, C. Sabnis, W. Sanders, R. Schantz, and M. Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–49, January 2003.
13. V. Roubtsov. My kingdom for a good timer! *Javaworld*, January 2003. `http://www.javaworld.com`.
14. S. Bagchi Z. Kalbarczyk, I. Ravishankar and K. Whisnant. Chameleon: A software infrastructure for adaptive fault-tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.