ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Reliable Communication

Deliverable Identifier:	D2
Delivery Date:	February 2004
Classification:	Public Circulation
Authors:	**Alberto Bartoli, Milan Prica, Etienne Antoniutti**
Contract Start Date:	1 September 2002
Duration:	36 months
Project coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università degli Studi di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd. (UK)

Project funded by the European Commission under the Information Society Technologies Programme of the 5th Framework (1998-2002)

CONTENTS

1. Introduction

The group communication system that we have designed and implemented as part of Workpackage WP1 (Task 1.2) is called JBora[1]. The JBora prototype can be downloaded from the following URL: http://adapt.ls.fi.upm.es/Downloads.htm.

In this document we discuss the main design decisions. Many of these decisions result from the design of several replicated services with strong consistency requirements that we performed recently in the framework of WP1 [Bartoli 2004, Bartoli and Kemme 2003, Bartoli *et al.* 2003]. Such designs allowed us to better understand the needs of replicated services deployed on top of group communication and meant to be accessed not only from local clients, but possibly from remote clients through the Internet.

JBora is fully implemented in Java. The API is documented in the prototype, along with a simple programming example and installation instructions. JBora consists of a Java layer on top of the Spread group communication toolkit (http://www.spread.org). Spread is a widely used and robust tool for group communication. It is also freely available, both in source form and in binary form. We purposefully used Spread as a black box: we did not modify Spread in any way and relied only on some of its basic functionalities. Spread is internally written in C but a Java interface is available. JBora relies on this Java interface.

2. Overview of group communication

We consider a distributed system modeled as a collection of processes that communicate through a network and that do not share storage. Processes may crash and communication failures may partition the network. A crashed process may recover as a “new” process and partitions are eventually repaired. The system is asynchronous in that no bounds are assumed on communication delays or relative speeds of processes. Message corruption and byzantine faults are excluded.

We consider services replicated across a collection of servers that form a *group*. Each server is equipped with a group communication middleware (GC-layer). Below we provide a brief overview of group communication to establish some background and refer to the abundant literature on this topic for a more rigorous description (e.g., [CACM 1996]). In the next section we will present the specific features of JBora, our GC-layer. We will use the terms “server” and “process” interchangeably.

A process joins the group with the grp.join() operation (we assume there is only one group for ease of presentation; grp is an instance of the JBora class). A process leaves the group either explicitly, with the grp.leave() operation, or implicitly, by failing. The grp.receive() operation is a blocking operation that returns either a message or a *view change*. A *view* is a set of process identifiers corresponding to the current group membership. A view change *vchg(v)* notifies the receiving process that the view has changed and has become *v*. The GC-layer determines a new view not only as a result of explicit join and leave operations, but also as a result of crashes, recoveries, network partitions and mergers. The GC-layer ensures that the perception of the group membership evolution at the various processes is “consistent”: all processes in *v* receive *vchg(v)* and view changes are received in the same order at all processes (see [CACM 1996] for a more rigorous description).

Group communication middleware is often based on a *primary-partition* model, where in case of network failures that split the group into two or more partitions, the system selects (at most) one such partition to represent the group as a whole and members that are in other partitions are forcibly expelled from the group [Ricciardi and Birman 1991, Kaashoek and Tanenbaum 1991, Mishra *et al.* 1991, Melliar-Smith *et al.* 1994, Malloth 1996]. A *partitionable* model is more general in that multiple “concurrent” partitions of the same group may exist and it is left up to the application to decide what may and may not be done in each partition [Amir *et al.* 1992, Dolev *et al.* 1995, Van Renesse *et al.* 1996, Babaoglu *et al.* 2001]. A partitionable framework is more appropriate for implementing replicated services since it delegates interpretation of partitioned operation semantics to the application rather than enforcing decisions at the group communication layer. And it is general enough to include the primary-partition model as a special case should applications require it [Babaoglu *et al.* 1997, Bartoli and Babaoglu 2003].

[1] The term bora indicates a very strong, cold wind that is typical of Trieste.

Group members communicate by means of grp.multiCast(m), that sends message *m* through a *totally-ordered* multicast: if processes *p* and *q* receive messages *m1* and *m2* then *p* and *q* receive these messages in the same order, i.e., either they both receive *m1* before *m2* or they both receive *m2* before *m1*.

The deliveries of multicasts and of view changes guarantee *uniform delivery* and *virtual synchrony*, as follows. We say that *p delivers view v* if *p* receives the associated view change *vchg(v)*. We say that *p delivers m in v* if the last view delivered by *p* before *m* is *v*:

- Let *p*,*q* deliver *v*. If *p* delivers *m* in *v*, then: (i) *q* also delivers *m*, unless *q* crashes; and (ii) if *q* delivers *m*, it does so in *v*.

These properties are very powerful for programming algorithms that have to cope with failures and recoveries. As a key example, consider a process *p* that delivers view *v*. Suppose that *p* crashes while multicasting *m* and, as a result, view *w* is delivered. It is guaranteed that either (i) *all* processes that deliver *v* and *w* receive *m* (and do so *before* receiving *w*); or (ii) *none* of them receives *m*. From a different point of view, when a process *q* delivers a message *m*, *q* can conclude that every other view member will deliver *m* as well, unless it crashes before.

Without uniform delivery, a process *q* that delivers a message *m* could not conclude anything about the delivery of *m*. Executions where *q* is the *only* process that delivers *m* would be allowed (such executions cannot occur if uniform delivery is guaranteed). A replication algorithm capable of coping with these executions correctly must include complex and costly reconfiguration procedures clients [Karamanolis and Magee 1999]. Such procedures are not needed when uniform delivery is guaranteed.

3. Specific features of JBora

JBora supports a partitionable membership model. Since most practical uses of group communication need a notion of “primary partition”, JBora augments each view with a flag telling whether that view is primary or not. Whether to exploit this information or to ignore it, is left to the upper layers.

Combining the notion of partitionable membership, primary view and uniform delivery introduces many subtle and difficult issues (Section 3.4). These issues are completely hidden to the upper layers thanks to the novel, simple semantics for message delivery that has been defined for JBora (Section 3.5).

Group composition is limited to servers only and clients do not join the group (Section 3.1). This is essential to guarantee scalability of the system with respect to large numbers of potential clients [Karamanolis and Magee 1999].

Each client interacts with a single server through a communication channel beyond the control of JBora, e.g., a TCP connection. This simplifies interoperability, allows integration with other middleware technologies, and services can be accessed by geographically-dispersed clients. Moreover, clients do not run locally any group communication protocol. All these features are fundamental for Internet-based services.

JBora makes no hypothesis about the transmission policy implemented by clients. For example, if a client does not receive a response to a request after some time (e.g., due to a broken TCP connection), it may resubmit the *same* request immediately to either the original server or some other one. As far as JBora is concerned, a client might even be connected to multiple replicas at the same time.

JBora includes a novel mechanism for propagating a load index among the replicas (Section 3.2). This mechanism, that we call *whiteboard*, is simple to use, cheap to implement and propagates the relevant information rapidly (a few msecs). Upper layers may implement load balancing policies based on the whiteboard mechanism. JBora need not know anything about such policies. The meaning of the load index is also irrelevant to JBora.

Finally, JBora incorporates a novel primitive that makes it very simple the propagation of relatively small pieces of state, in order to simplify the implementation of state transfer procedures (Section 3.3).

3.1. Clients that are not group members

Each client communicates with a single server through a communication channel beyond the control of JBora, e.g., a TCP connection. Servers are typically multi-threaded. It is often the case, for example in servlet containers and application servers, that each request is processed in the context of a separate thread.

As a result, each server has to handle multiple, independent flows of events: requests from clients, each client being associated with a dedicated connection; connection requests from new clients; messages and view changes from the GC layer. In practice, coordinating these flows may be a significant source of complexity.

We paid special attention to this issue, that we solved as follows. We expect that a thread in the server will be organized as grp.receive() loop. Although this is not mandatory, servers are typically structured this way. The main reason is because this structuring makes it simpler to exploit the ordering guarantees provided by the GC layer (these guarantees are defined per-process, not per-thread). We devised two novel calls for inter-thread communication to be exported by the GC layer. These novel calls are localCall() and localRespond(). resp←grp.localCall(msg) sends msg to the executing process, that is, it inserts msg in the flow of events that the GC layer delivers to the executing process; moreover, this primitive blocks the invoking thread T1; note, msg will be received by another thread T2, through grp.receive(). Thread T1 unblocks when T2 executes grp.localRespond(msg,resp).

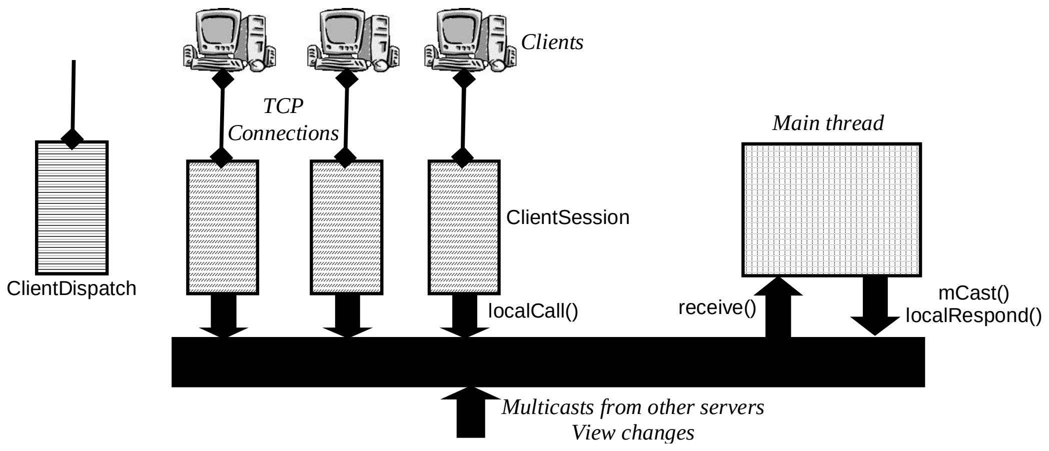

Figure 1 Example of threading architecture for a server (boxes indicate threads)

The above figure shows a possible threading architecture for a server. When the server bootstraps, it spawns a MainThread thread and a ClientDispatch thread. The former is the only thread that multicasts messages and listens to messages and view changes. The latter associates a ClientSession thread with each connection. The ClientSession thread could have the same lifetime as the connection, or it could be allocated from a pool when the connection is created and returned to the pool when the connection is closed. When a ClientSession processes a request involving a multicast, the ClientSession executes the JBora localCall() primitive. This action inserts the request into the flow of events processed by the main thread. When the request has been processed, the main thread will execute localRespond() thereby awakening the corresponding ClientSession. Note, when the MainThread is blocked on a JBora receive(), it may be awakened by either a multicast, or a view change, or a message sent with a localCall(). Note also that mCast() could be issued by any thread, not necessarily by the MainThread.

Servlet containers and application servers require a threading architecture slightly different from the one of the above example. The only difference is that TCP connections, as well as the association between threads and requests, are fully managed by the container. Everything else, including the use of JBora primitives, remains unchanged. A detailed example for these environments can be found in [Bartoli *et al.* 2003].

3.2. Whiteboard

Each group member has access to a grp.WhiteBoard object that it can read and write. The collection of these objects is meant to simulate a "whiteboard" shared among the group members. Each grp.WhiteBoard instance is a table of *integers*, with one element corresponding to each member (see also the end of this section for a discussion about the type of whiteboard elements). The call grp.WhiteBoard.putValue(val) sets the element of the table corresponding to the invoking member to val. The call grp.WhiteBoard.getValue(p) returns the value of the table element associated with group member p.

Writes are *not* propagated with virtual synchrony semantics: if, for example, *p* applies a sequence *X* of writes to grp.WhiteBoard while in view *v*, then different members of *v* could observe different subsequences of *X*. Causal precedence relationships between writes to the grp.WhiteBoard and multicast transmissions may not be preserved either: if, for example, *p* issues grp.WhiteBoard.Write(val) and then multicasts *m*, then different processes could observe the write and the delivery of *m* in different orders.

The semantics associated with grp.WhiteBoard have been purposefully kept weak in order to admit cheap implementations. In our implementation, the table is replicated at each group member. Reads are performed on the local copy, whereas writes are performed on the local copy and then propagated to other copies by piggybacking the new value to messages that the GC layer has to exchange anyway (multicasts, view changes). It follows that writes propagate at basically no cost and, typically, within a few hundreds of msecs.

The whiteboard is useful for propagating a load index among the replicas. A detailed example, including a performance evaluation, can be found in [Bartoli *et al.* 2003]. The example uses the number of *in-progress requests* as load index and builds a *threshold-based* load balancing policy based upon this index. We remark again that updates to the whiteboard need *not* trigger additional multicasts: the whiteboard value is automatically piggybacked into messages that the GC-layer has to exchange anyway.

Of course, one could define many different policies based on other load indices — e.g., average CPU load, number of open connections, number of requests processed in the last *k* seconds, *k* being a configurable parameter. The nature of the load index is irrelevant to the whiteboard mechanism.

The reason why we designed the whiteboard as a table of integers rather than a more general table of `Object`s, is performance. The whiteboard should be very small, and it should be serialized and deserialized very quickly and very efficiently. While adding a bunch of integers to each group message can be done at basically no cost, handling more complex objects would certainly add some fixed and not negligible overhead to JBora. It could be possible, though, to extend the whiteboard mechanism so as to support the association of several integers with each group member.

3.3. State transfer

An issue that has to be addressed by every application deployed on top of group communication is the propagation of application-defined information upon a *view expansion.* When a replica recovers after a failure or joins the group for the first time, the state of the replicated service has to be made known to the replica. This operation is called *state transfer*. The topic is discussed in depth in [Babaoglu *et al.* 1997, Kemme *et al.* 2001].

JBora incorporates a novel primitive, called propagate(), that makes it very simple the propagation of relatively small pieces of state — up to a few KBytes. In practice, a service will perform state transfer by means of a mixed approach. First, the propagate() primitive will be invoked upon a view expansion for transferring the basic portions of the replicated service state (e.g., the identity of the "primary" server). Then, another application-specific approach will be used for transferring the larger portions of the replicated service state (e.g., all active HttpSession objects, in a service that replicates session objects; or the actual database state, in a replicated database). We decided not to include in JBora more sophisticated support for state transfer, because some of the replication algorithms that are being exploited in WP1 do not need that all replicas in the same view always store the very same state. For these algorithms, the propagate() primitive should provide all that is actually needed.

The primitive propagate(msg) is a blocking operation that multicasts msg and terminates upon receiving a message sent through propagate() from each member of the view. The operation returns: (i) a table stTable with one element per view member, containing the message sent by that member; and (ii) a list evList of the events delivered while waiting for completion (messages not sent with propagate(), view changes).

The idea is that a process *p* should invoke propagate(msg), where msg describes the state of *p*, whenever *p* delivers a view expansion. When the invocation completes, *p* will have locally available the state of all other view members (within the stTable). In other words, the execution of propagate() within JBora is similar to a loop that executes receive() and terminates upon receiving a message from each view member.

To realize the power of propagate(), suppose that this primitive is not available. That is, when a process *p* delivers a view expansion, *p* invokes mCast(msg) instead than propagate(msg). Process p will expect to receive a message from each member of the new view. Unfortunately, while waiting for these messages, *p* could deliver further view changes (contractions and/or expansions) as well as other application-level messages (e.g., messages from clients of the replicated service). Taking care of all the relevant details is, in practice, annoying and a significant source of complexity. The semantics of propagate() have been carefully defined so as to encapsulate most of these low-level details, that otherwise would have to be programmed explicitly.

3.4. Partitionable Membership, Primary View, Uniform Delivery (I)

Combining the notion of partitionable membership, primary view and uniform delivery introduces many subtle and difficult issues. In this section we elaborate on this claim.

Given two views v and w, we say that: (i) w is an *immediate successor* of v (or v is an *immediate predecessor* of w) iff there is a process for which w is the next view to be delivered after v; (ii) w is a *successor* of v (or v is a *predecessor* of w) iff there is a sequence of views $v, v_1, \ldots v_k, w$ ($k \geq 0$) such that each view is an immediate predecessor of the next view in the sequence; (iii) w and v are *concurrent* iff neither view is a successor of the other. Concurrent views are views that are installed at different processes and reflect different perceptions of the group membership, typically as a result of partitions (Figure 2a).

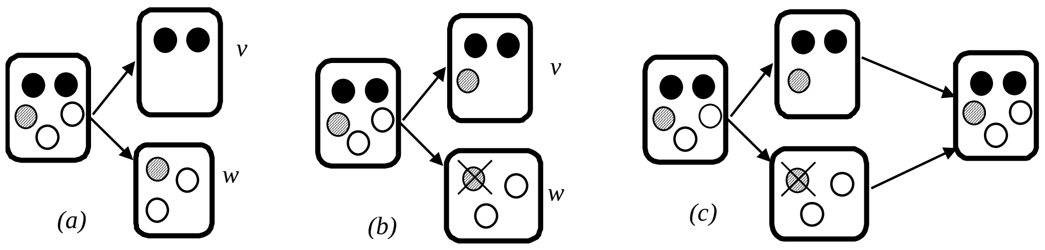

Figure 2. (a,b) Examples of concurrent views (heavy outline boxes indicate views, circles indicate processes, an arrow from a view to another indicates that the former is an immediate predecessor of the latter). (c) An execution that is impossible since merging views *v* and *w* do not have an empty intersection.

Many applications need a notion of *primary view*. This means that primary views must form a total order: given two views *v* and *w* that are both primary views, *v* and *w* cannot be concurrent, i.e., one must be a successor of the other. The notion of primary view is typically implemented based on either *majority* of *dynamic voting*. In the former case, a view is a primary view iff it contains a strict majority of a statically known set of replicas. With dynamic voting, the first primary view, say v_0, has a predefined composition. Then, view *v* is primary iff (i) one of one of its immediate predecessors, say *w*, is primary, and (ii) *v* includes a strict majority of the members of *w*.

In principle, one could implement one of the above criteria either at the application level or in the GC-layer itself. Unfortunately, there are many subtle details that make it very difficult the job of implementing the notion of primary view in a partitionable system. We elaborate on this issue in the following.

One would implement either of the above criteria based on the composition of each view. But *concurrent views may have overlapping composition* (see below). It follows that one might end up with multiple concurrent views that all believe to be the primary view. This is unsafe, because by definition of primary view, any set of concurrent views should contain at most one primary view.

As an example of concurrent views with non-empty intersection, a process *p* might be a member of view *w* even though *p* did not install *w* and installed a different view *v*, concurrent to *w* (Figure 2b; the crossed out process is a member of *w* even though it installed *v*). The reason why this can occur is beyond the scope of this document. It suffices to observe that this possibility cannot be avoided [Babaoglu *et al*. 1995], unless one constructs a membership service that might block in case of failures occurring during a view change protocol — clearly an undesirable design choice. It is guaranteed that concurrent views having a common immediate successor have empty intersection (Figure 2c). This means, in particular, views that have not be installed by all of their members will disappear and be followed by views not including such members.

Consider a process *p* that delivers a view containing a majority of group members (the reasoning is unchanged if dynamic voting is used). The process cannot tell, *based solely on the view composition*, whether other concurrent views also contain a majority of group members. For example, suppose the circles in Figure 2 denote all group members. White processes cannot distinguish between Figure 2a and Figure 2b based solely on the composition of *w*. Thus, they cannot tell whether it is safe to qualify *w* as a primary view.

In addition to tell safely whether a view is primary, another difficulty is related to uniform delivery. The semantics of uniform delivery in partitionable systems has been formalized in the *Extended Virtual Synchrony* model [Moser *et al.* 1994]. The cited work shows that such semantics can only be quite complex. We briefly outline this complexity in the following and refer to the cited work for the precise specification.

Let *v.comp* denote the composition of view *v*. If a process *p* delivers a message *m* in view *v*, then *p* can conclude that each process $q \in v.comp$ either delivers *m* or crashes before doing so. However, *p* may also deliver *m* along with a description of the subset of *v.comp* that *might not* deliver *m* before crashing. In this case we say that *p* delivers *m* as an *in-doubt* message. Since in-doubt messages cannot be avoided, applications must be prepared to cope with them. The clean and simple semantics of uniform delivery is thus lost and the design of algorithms becomes much more complex. The description of the subset of *v.comp* that might not deliver an in-doubt message takes the form of a transitional view. That is, views may be delivered either as *regular* views or as *transitional* views. In-doubt messages may result from network partitions occurring while *m* is being multicast. To complicate things further, a message might be delivered as in-doubt at some processes but not at other processes.

3.5. Partitionable Membership, Primary View, Uniform Delivery (II)

In this section we describe how the issues presented in the previous section are completely hidden to the upper layers thanks to the new, simple semantics that has been defined and implemented in JBora. No other group communication system provides this semantics.

JBora marks each delivered view with a flag telling whether the view is primary or not. That is, the upper layers implement predicate isPrimary(v) by simply inspecting the value of the flag associated with *v*. JBora selects the value for this flag by means of a majority-based rule.

The evaluation of isPrimary(v) may return true at processes in concurrent views, but this is harmless to the application because of the delivery properties discussed below. These properties ensure that, given any set of concurrent views, messages can be delivered in at most one view in the set. This view is the "real" primary view. Other views possibly marked as primary will disappear soon (guaranteed by the group communication layer) without delivering any message in the view (guaranteed by JBora).

The solution resides in the JBora primitive multiCast(msg). This primitive ensures the following property that we call "Prefix Delivery"[2]. Let predicate isPrimary(v) be true iff *v* is a primary view. Let processes *p* and *q* be two members of view *v* and let isPrimary(v)=true. Let, finally, *p* receive the next view w_p and let *q* receive the next view w_q. The property is as follows:

- If $w_p = w_q \Rightarrow p$ and *q* receive the same sequence of messages in v.
- Else if $w_p \neq w_q$ and *isPrimary*$(w_p) \Rightarrow$ *q* receives in *v* a prefix of the sequence or the same sequence of messages received by *p* in *v*.
- Else if $w_p \neq w_q$ and *isPrimary*(w_p) and *isPrimary*$(w_q) \Rightarrow$ either *q* and *p* deliver the same sequence of messages in *v* or one delivers a prefix of the sequence delivered by the other.

Essentially, sites that continue to be in the primary view must not miss any message. The Prefix Delivery property includes the virtual synchrony property and prevents the occurrence of "holes" in the total order such as the following: *p* receives the sequence of messages m_1, m_2, m_3 while *q* only receives m_1, m_3. This semantics is very powerful yet simple to understand.

4. Implementation outline

We decided to leverage on existing group communication middleware rather than to develop our own. Developing a group communication system from scratch would require a considerable effort and would involve tackling many subtle problems, both from the point of view of correctness and of performance. The effort would be largely orthogonal to the main objectives of Adapt.

The two platforms closer to our needs were Spread (http://www.spread.org) and JavaGroups (http://www.javagroups.org). Both platforms are widely used, robust and freely available. We selected Spread because, unlike JavaGroups, it implements completely the Extended Virtual Synchrony model, including full support for partitionable operation and uniform delivery (called *safe* delivery in that model). These features should definitely be available to the replication algorithms of interest in Adapt. Although JavaGroups can be easily augmented with additional features, implementing the Extended Virtual Synchrony model would be very complex. Spread is written in C and distributed in binary form, thus it offers much better performance than JavaGroups, in particular under high load (see also the Adapt Deliverable D2, Basic Services Architecture). We purposefully used Spread as a black box: we did not modify Spread in any way and relied only on some of its basic functionalities. Spread is internally written in C but a Java interface is available. JBora relies on this Java interface.

JBora implements all the features discussed in Section 3, none of which was available in Spread.

The whiteboard is implemented as a table replicated at each process. Reads are performed on the local copy. Updates are performed on the local copy and then piggybacked within messages that are multicast anyway. Each multicast includes a small JBora-specific header. This header carries the updates to the whiteboard possibly triggered by the sender of the multicast (recall that a process can only update the entry in the table associated with itself). The serialization machinery within JBora ensures that the header is inserted without any additional memory-to-memory copy (i.e., the object to be multicast is serialized after the JBora-specific header). To make sure that updates to the whiteboard are propagated even when the process does not generate any multicast, an empty multicast is generated every now and then, usually with a period of a few hundred msecs.

Events delivered by Spread (messages, transitional views, regular views) are analyzed by a thread within JBora. For each event this thread, called spreadJBora, decides whether to pass the event up to the JBora interface, to drop it, or to buffer it. Dropping occurs, for example, if the event is a transitional view (transitional views are not exposed at the JBora interface). Buffering occurs, for example, when a message is delivered in transitional view; buffered messages will be either delivered or discarded when Spread delivers the next regular view; the choice between deliver and discard is done so as to guarantee the Prefix Delivery property (Section 3.5).

Events passed up to the application are queued. The receive() operation extracts one event from this queue. The localCall(m) primitive for inter-thread communication is implemented by inserting *m* into

[2] This property is the result of joint work with Bettina Kemme (see also [Bartoli and Kemme 2003]). It constitutes a more compact and more rigorous description of earlier attempts that we made during the design.

this queue, i.e., in the flow of events to be delivered to the application. The thread that executes localCall() is suspended and then awakened (upon execution of localRespond() by another thread) by means of common concurrent programming techniques.

The propagate(m) operation is implemented by multicasting m along with the identifier of the last view delivered to the upper layers (this identifier is unique for each view system-wide; it is generated by Spread and is not exposed to the upper layers). Then, the receive() operation is executed within a loop that exits as soon as a propagate message has been received from every view member. If a view change occurs before exiting the loop, then propagate messages already received are discarded and the procedure restarts. Propagate messages not carrying the identifier of the last view delivered to the upper layer are discarded.

5. Adaptive message packing

The powerful guarantees provided by group communication greatly simplify the job of implementing replication algorithms that behave correctly in spite of failures and recoveries. But these guarantees have a significant run-time cost. As an aside, we point out that JBora adds little overhead to Spread: JBora sustains a throughput only slightly smaller than Spread. We decided to investigate techniques for improving the efficiency of the group communication system. These techniques, that we call *adaptive message packing* and are described in this section, appear to be quite promising but their effectiveness in a complete replication algorithm (e.g., in a J2EE environment) is not yet fully clear. Accordingly, we decided to deliver a prototype of JBora that does not incorporate adaptive message packing. We will continue investigating this optimization in the context of Task 1.1 (Architecture and Implementation of BS middleware). We will decide at a later stage whether to deliver a further JBora prototype including adaptive message packing. The new prototype will not affect code already developed above JBora in any way.

5.1. Message packing[3]

In our test environment, 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 (total order and safe, also called uniform, 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 message per operation, it is easy to see that group communication may potentially 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. The starting point of our work is the proposal by Friedman and Van Renesse [Friedman and Van Renesse 1997], in which they demonstrated that message packing could 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 (a similar technique is used in the implementation of many networking protocols, e.g., TCP). 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 [Friedman and Van Renesse 1997]. 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

[3] This section is an excerpt from *"Adaptive Message Packing for Group Communication Systems"*, A. Bartoli, C. Calabrese, M. Prica, E. Antoniutti Di Muro, A. Montresor, Workshop on Reliable and Secure Middleware, pp. 912-925, Lecture Notes on Computer Science 2889, Springer Verlag, November 2003.

could even stop sending messages. 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 following). 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.

5.2. Operating environment and measurements

Our group communication system, JBora, consists of a harness around 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 multiCast(m). Rather than invoking the multicast operation of Spread immediately, multiCast(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 dynamically based on observed execution statistics (see Section 5.3).

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. In the experiments discussed here, this was not an issue.

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 focused 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 focused 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 [Friedman and Van Renesse 1997] 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 [Karamanolis and Magee 1999]. Second, we considered sources that generate messages continuously or at bursts (alternating a burst with a sleeping time). That is, unlike [Friedman and Van Renesse 1997], 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).

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.

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 [Roubtsov 2003].

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.

5.3. 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 3. 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 3 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 Concluding remarks).

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 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.

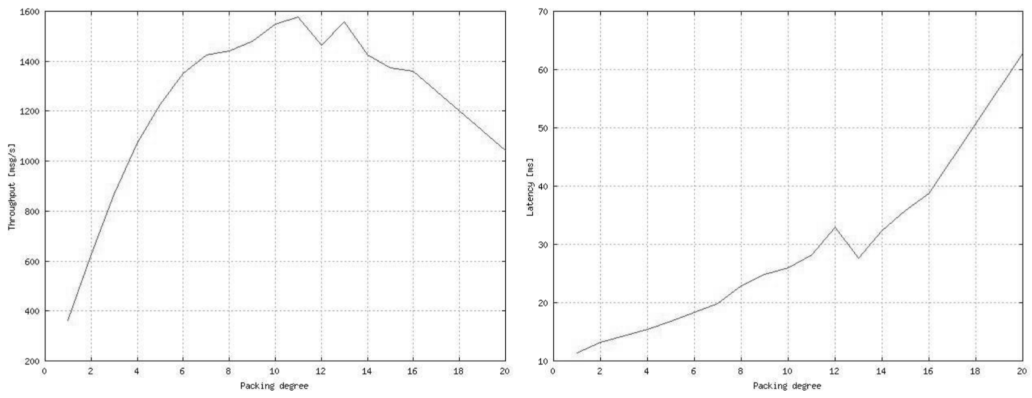

Figure 3 Throughput (left) and latency (right) with varying packing degrees (msg size is 1000 bytes)

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):

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

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

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

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 4. Suppose the system be characterized by curve B and the packing degree has reached its optimal value p_B. 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 p_B-1. The next measure will then show that throughput has increased, thus *pack* will be incremented again at p_B. Since this increment will cause throughput to decrease, at this point the value of *pack* will keep on oscillating around p_B, 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:

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

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.

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 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).

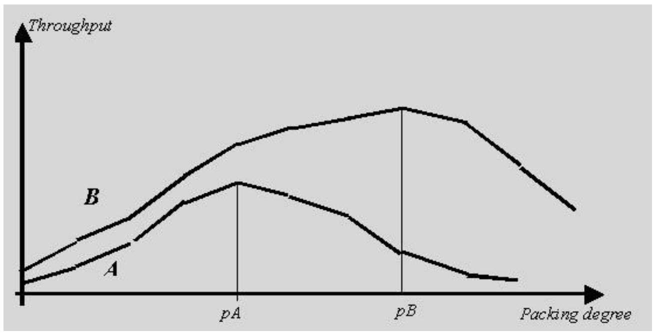

Figure 4 Throughput vs Packing degree for different loads: curve A corresponds to an higher load than curve B (curves for different message sizes and/or additional load have the same shape).

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 300000 1000-byte messages followed by 300000 3000-byte messages and then repeats this pattern indefinitely.

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 5-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 5-right shows the variations of *pack* over a short time interval.

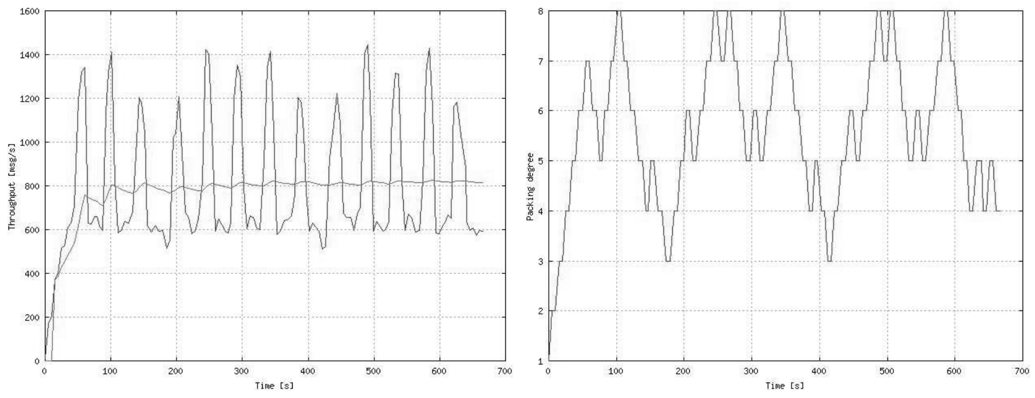

Figure 5 Average throughput and "instantaneous" throughput over time (left). Packing degree over time (right).

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.

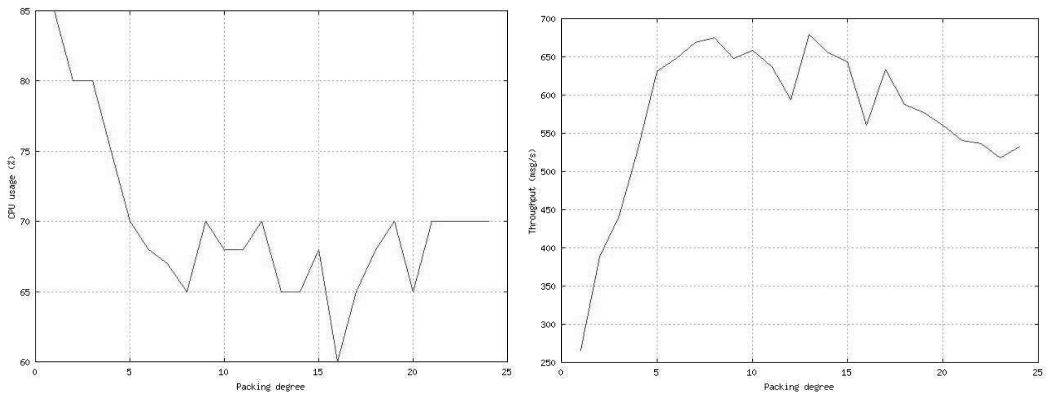

Figure 6 Average CPU usage (left) and average throughput (right) over packing degree. Bursty source; 15 messages and then 20 msecs sleeping time. Note that the average CPU usage is plotted in the range 60%-85%, not in the full 0-100% range.

These experiments show 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 6. Without packing enabled, CPU usage is 85% and throughput is approximately 260 msg/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 6-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%.

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 msg/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.

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.

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.

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.

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. The main drawback of message packing is that it tends to increase latency. Yet we believe that the resulting trade-off between throughput, CPU usage, latency that is enabled by our adaptive policy for message packing is worth exploring.

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 \textit{packing interval} $\ptime$. 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). XXX

As pointed out at the begininning of this section, we intend to continue investigating adaptive message packing in the context of Task 1.1 and we will decide at at a later stage whether to deliver a further JBora prototype including this optimization. The new JBora prototype that we are developing based on the results reported here will trigger transmission when either of the following conditions holds:

1. One of the messages has been in the buffer for a time larger than a predefined quantity, say $time_P$.
2. The aggregate size of the buffered messages is larger than a predefined quantity, say $size_P$.

Condition 1 is for handling very irregular sources without introducing unacceptably high latency times within the packing buffer (as well as for handling sources that could not even fill the packing buffer). Condition 2 is because realistic sources will not generate equally-sized messages. Quantity $size_P$ is essentially equivalent to the packing degree *pack* that we have investigated, in that it is the quantity that will be determined automatically by our policy (the only difference being the amount of each update, that was 1 in our experiments and could be 250 bytes or so for $size_P$). Quantity $time_P$ could be also adjusted dynamically, in principle. However, we believe that dynamically adapting both $time_P$ and $size_P$ could be exceedingly difficult in practice. We believe it is more practical to define $time_P$ statically based on an estimated upper bound for the time spent in the packing buffer, which can be derived from the latency requirements of the application.

References

[Amir *et al.* 1992] Y. Amir, D. Dolev, S. Kramer, D. Malkhi "Transis: A communication sub-system for high availability", *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pp. 76-84, July 1992

[Babaoglu *et al*. 1995] Ö. Babaoglu, R. Davoli, L. Giachini, P. Sabattini, "The inherent cost of strong-partial view synchronous communication", *in Distributed Algorithms (WDAG9), Lecture Notes in Computer Science 972*, October 1995, pp.72-86.

[Babaoglu *et al*. 1997] Ö. Babaoglu, A. Bartoli, G. Dini, "Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems", *IEEE Transactions on Computers*, 46 (6) pp. 642-658, June 1997.

[Babaoglu *et al*. 2001] Ö. Babaoglu, R. Davoli, A. Montresor, "Group Communication in Partitionable Systems: Specification and Algorithms", *IEEE Transactions on Software Engineering 27(4):308-336, April 2001.*

[Bartoli 2004] A. Bartoli, "Implementing a Replicated Service with Group Communication", Journal of Systems Architecture, Elsevier, to appear.

[Bartoli and Kemme 2003] A. Bartoli, B. Kemme "Recovering from Total Failures in Replicated Databases", Technical Report 2003, Available from http://webdeei.univ.trieste.it/Archivio/Docenti/Bartoli/repExt.pdf

[Bartoli *et al.* 2003] A. Bartoli, M. Prica, E. Antoniutti "A Replication Framework for Program-to-Program Interaction across Unreliable Networks and its Implementation in a Servlet Container", Technical Report 2003, Submitted for publication, Available from http://webdeei.univ.trieste.it/Archivio/Docenti/Bartoli/repProgram.pdf

[Bartoli and Babaoglu 2003] A. Bartoli, O. Babaoglu, "Application-Based Dynamic Primary Views in Asynchronous Distributed Systems", Journal of Parallel and Distributed Computing, vol. 63 (2003), pp. 410-433.

[CACM 1996] Special Issue on Group Communication, *Communications of the ACM*, 39(4), April 1996.

[Dolev *et al*. 1995] D. Dolev, D. Malkhi, R. Strong, "A Framework for Partitionable Membership Service", Technical Report 95-4, Dept. of Computer Science, The Hebrew University of Jerusalem, 1995.

[Friedman and van Renesse 1997] R. Friedman, R. van Renesse, “Packing messages as a tool for boosting the performance of total ordering protocols”, *Proc. of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC ’97)*, 1997.

[Kaashoek and Tanenbaum 1991] F. Kaashoek, A. Tanenbaum, "Group communication in the Amoeba distributed operating system", *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, pp. 222-230, May 1991.

[Karamanolis and Magee 1999] C. Karamanolis and J. Magee, "Client-access Protocols for Replicated Services", *IEEE Transactions on Software Engineering*, 25 (1), pp.3-21, February 1999.

[Kemme *et al.* 2001] B. Kemme, A. Bartoli, Ö. Babaoglu, “On-line reconfiguration in replicated databases based on group communication”, *Proc. IEEE/IFIP Dependable Systems and Networks 2001*, pp. 117-126.

[Malloth 1996] C. Malloth, “Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks”, Ph.D. Thesis, EPFL, Lausanne 1996.

[Melliar-Smith *et al*. 1994] P. Melliar-Smith, L. Moser, V. Agrawala, "Processor Membership in Asynchronous Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, 5 (5) pp. 459-473, May 1994.

[Mishra *et al*. 1991] S. Mishra, L. Peterson, R. Schlichting, “A membership protocol based on partial order”, Proceedings of the International Workshop on Parallel and Distributed Algorithms, pp. 137-145, February 1991.

[Moser *et al*. 1994] L.E. Moser, Y. Amir, P.M. Melliar-Smith, D.A. Agarwal, "Extended Virtual Synchrony" *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 56-65, June 1994.

[Ricciardi and Birman 1991] A. Ricciardi, K. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments", in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pp. 341-352, August 1991.

[Roubtsov 2003] V. Roubtsov, “My Kingdom for a Good Timer”, *Javaworld*, January 1993.

[Van Renesse *et al*. 1996] R. Van Renesse, K. Birman, S. Maffeis, "Horus: A Flexible Group Communication System", *Communications of the ACM*, 39 (4) pp. 76-83, April 1996.