On the Cost of Modularity in Atomic Broadcast

Olivier Rütti[†] olivier.rutti@epfl.ch
Sergio Mena[‡] sergio.mena@cs.york.ac.uk
Richard Ekwall[†] nilsrichard.ekwall@epfl.ch
André Schiper[†] andre.schiper@epfl.ch

[†] *École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland*
[‡]*Department of Computer Science, University of York, York YO10 5DD, United Kingdom*

Abstract

Modularity is a desirable property of complex software systems, since it simplifies code reuse, verification, maintenance, etc. However, the use of loosely coupled modules introduces a performance overhead. This overhead is often considered negligible, but this is not always the case. This paper aims at casting some light on the cost, in terms of performance, that is incurred when designing a relevant group communication protocol with modularity in mind: atomic broadcast.

We conduct our experiments using two versions of atomic broadcast: a modular version and a monolithic one. We then measure the performance of both implementations under different system loads. Our results show that the overhead introduced by modularity is strongly related to the level of stress to which the system is subjected, and in the worst cases, reaches approximately 50%.

Keywords: atomic broadcast, modular design, microprotocols, performance cost, experimental evaluation

1 Introduction

Modularity has always been an important concern when designing complex software systems. A modular system is easier to maintain, its code being easier to debug, verify, reuse and develop in a collaborative environment. However, modularity is not a panacea and its main drawback is the performance overhead introduced by splitting the system into several independent parts. Such overhead is often deemed negligible when compared to all the good properties that modularity entails; but it is usually difficult to perform a quantitative analysis of the actual performance impact.

Group communication has been argued to be an important enabling technology to render a distributed service fault-tolerant by replicating such service at several locations [5, 8]. In this context, atomic broadcast is a well-known protocol that allows to maintain replicas consistency by ensuring a total order of message delivery. In [13, 7], Chandra and Toueg propose a reduction of this protocol to the consensus problem. This allows a modular design of atomic broadcast based on consensus and reliable broadcast. In such a design, atomic broadcast knows that it is interacting with a consensus module, but cannot make any assumption on the implementation of the consensus module (e.g., which algorithm is used). As a result, some algorithmic optimizations that make assumptions on the neighbor protocol can not take place if the system is to be modular: this is where the performance penalty is mostly located.

Is it not so easy to decide between a modular design or a monolithic one: this decision has to be made at the early stages of the software engineering process, whereas evidence of the performance cost can only be obtained later, when at least a prototype is available. Nevertheless, it is possible to foresee the performance hit at design time using an analytical method (See Sect. 5.2).

Contribution. This paper aims at shedding some light on the performance cost that modularity induces in implementation of atomic broadcast reduced to the consensus problem. For our experiments, we use Fortika [18, 19], a toolkit that includes two versions of atomic broadcast: monolithic and modular. Both versions are based on the same algorithms. In one version atomic broadcast is implemented as a set of modules, whereas in the other, these modules are merged to form a monolithic protocol. This merging allows algorithmic optimizations, since we can assume that these modules always operate together. Those optimizations aim at (a) improving the performance in *good runs* (runs where messages are timely and processes behave correctly[1]), and (b) keep algorithmic correctness in all runs. For a fair comparison, we also optimize modular version of atomic broadcast.

[1] Good runs are the most frequent in practice

37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)
0-7695-2855-4/07 $20.00 © 2007 IEEE

The performance of both modular and monolithic solutions are then shown in both analytical and experimental evaluations of the two stacks.[2] Our results reveal that the performance hit can reach 50% in some cases, showing that the dilemma between a monolithic and a modular design should not be taken lightly.

2 Atomic Broadcast

This section briefly presents the system model that we consider and concisely describes the modules that constitute the atomic broadcast stack.

2.1 System Model

We consider a system with a finite set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. The system is *asynchronous*, which means that there is no assumption on message transmission delays or relative speed of processes. The system is *static*, which means that the set Π of processes never changes after system start-up time. During system lifetime, processes can take internal steps or communicate by message exchange.

Correct, Faulty and Failure Suspicion. Processes can only fail by crashing. A process that crashes stops its operation permanently and never recovers. A process is *faulty* in a given run if it crashes in that run. A process is *correct* if it is not faulty. Every process has a local module called *failure detector* (FD) that outputs a set of processes that have crashed. This list can change over time, moreover it can be inaccurate. We say that process p *suspects* process q if q is in the output list of p's FD.

Quasi-Reliable Communication Channels. Every pair of processes is connected by a bidirectional network channel. The protocols presented later on assume *quasi-reliable channels*, which verify the following property. If process p sends message m to q, and both p and q are correct, then q eventually receives m.

2.2 Description of Modules

Our atomic broadcast implementation consists of three main protocols that are based on well-established algorithms: reliable broadcast, consensus and atomic broadcast. We now give a concise description of these protocols (see [13] for further details and formal specifications).

[2] We use the terms "stack" and "implementation" interchangeably

Reliable Broadcast. This protocol defines the primitives *rbcast* and *rdeliver*. Reliable broadcast ensures that messages are rdelivered either by all correct processes or by none, even if the sender crashes while rbcasting a message. However, it does not enforce any order in rdelivered messages.

Consensus. Consensus defines the primitives *propose* and *decide*, which mark the protocol's start and end at a given process. Consensus ensures that processes eventually reach an agreement on a value proposed by one of them, even in the presence of crashes.

Atomic Broadcast. This protocol defines the primitives *abcast* and *adeliver*. Atomic broadcast is a stronger form of reliable broadcast where all messages are adelivered in the same order at every process.

3 Modular Implementation

The current section describes the modular implementation of atomic broadcast (see Fig. 1, left). We present the implementation of all modules following a bottom-up order. These modules implement the protocols described in Sect. 2.2. Detailed knowledge of these implementations is not necessary to keep up with the rest of the paper. However, a succinct description will help the reader to better understand (1) the monolithic implementation presented in Sect. 4 and (2) the analytical evaluation presented in Sect. 5.2.

For each module, we present some optimizations that focus on good runs (runs with no suspicion, crash or unusual message delay). Our optimizations, however, do not affect the correctness of the algorithms in runs that are not good. These improvements are necessary to obtain a comparison as fair as possible, between the modular and monolithic stacks.

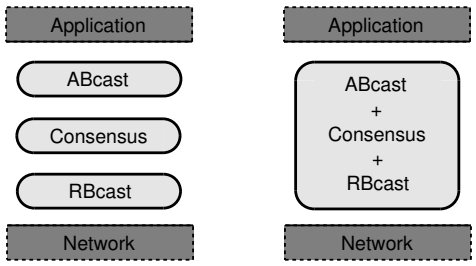

Figure 1. Modular implementation(left) and monolithic implementation(right) of Atomic Broadcast (ABcast).

0-7695-2855-4/07 $20.00

3.1 Reliable Broadcast (RBcast)

The classical implementation of this protocol is straightforward if we can assume quasi-reliable channels (see Sect. 2.1). Here is the main idea [7]:

1. Upon broadcast of message m, send a copy of m to all processes.
2. Upon receiving m for the first time, re-send m to all processes.

Optimization. Note that this implementation sends n^2 messages over the network for each rbcast message (n denotes the number of processes to which the message is broadcast). This can be reduced by assuming that a majority of processes do not crash[3]. This optimization leads to only $(n-1)\cdot(\lfloor\frac{n-1}{2}\rfloor+1)$ messages per rbcast message. The details of this optimization are omitted here.

3.2 Consensus

We base our implementation on the Chandra and Toueg consensus algorithm [7] due to its overall good performances [25]. Rather than presenting the full algorithm's details, we explain its principles by using a typical run, depicted in Fig. 2. The algorithm proceeds in a number of asynchronous rounds. In each round, a different process adopts the role of coordinator. A round consists of four phases:

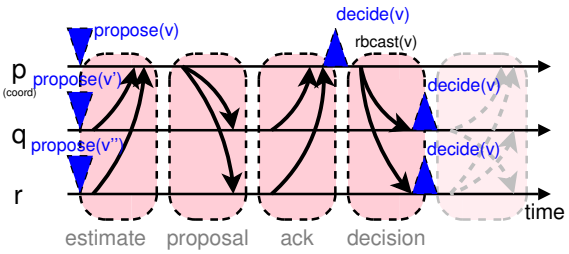

Figure 2. Example execution of consensus during good runs.

1. *Estimate* phase. All processes send their initial value as estimate to the coordinator.
2. *Propose* phase. The coordinator chooses the eldest value and sends a *propose* message with such value.
3. *Ack* phase. All processes wait for the coordinator's proposal and send an *ack* message when they receive it, or a *nack* message if they suspect the coordinator.
4. *Decide* phase. If the coordinator gathers *ack* messages from a majority of processes, it decides and rbcasts the decision to all processes. The last phase in Fig. 2 (grayed) is the re-send part of rbcast algorithm (see reliable broadcast implementation above in this section).

If the coordinator is faulty and/or suspected, the algorithm may not be able to decide in the first round. In that case, supplementary rounds with the same phases would be needed in order to terminate. At any moment, if a running process receives a decision, it decides the received value and terminates. In runs where there are no crashes or suspicions, all processes are able to decide at the end of the first round (see dark upward triangles in Fig. 2).

Optimization. Figure 3 shows a typical run of the consensus algorithm that we implemented. Firstly, we reduce the first round by suppressing the estimate phase. Secondly, contrary to classical implementation where round n+1 begins immediately after round n terminates, a new round starts only if the coordinator is suspected to be faulty. These two improvements were previously described in [25]. Finally, we reduce the size of decision messages by sending a tag DECISION instead of the complete decision. Note that, even if this optimization works fine in good runs, additional communication steps may be required if the coordinator crashes.

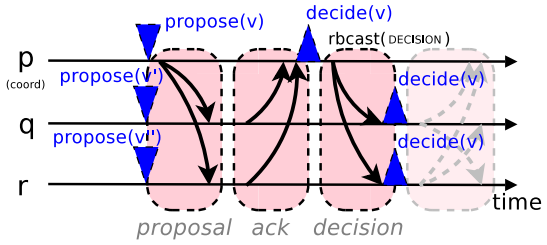

Figure 3. Example execution of optimized consensus during good runs.

3.3 Atomic Broadcast (ABcast)

We solve atomic broadcast by reduction to consensus [13, 7]. In this approach, the atomic broadcast module diffuses all messages abcast by the application. In parallel, a consensus is started to decide on the delivery order of those messages. Hence, consensus accepts a batch of messages as initial values. Figure 4 depicts an example execution where messages m (abcast by p's application) and m' (abcast by r's application) are abcast. First, both messages are disseminated to all processes; then, an instance of consensus is started to order m and m' consistently at all processes. When consensus decides, atomic broadcast

[3] The same assumption is necessary to solve consensus

0-7695-2855-4/07 $20.00

adelivers the messages contained in the decision in some deterministic order. In Fig. 4 for instance, m' happens to be ordered before m, but this order is consistent everywhere. Finally, the whole mechanism is repeated as soon as further messages are abcast.

Optimization. Note that in [13, 7], reliable broadcast is used to disseminate the messages abcast by the application. In our stack, messages are simply sent using quasi-reliable channels (solid arrows in Fig. 4). This implementation is clearly equivalent to reliable broadcast when no process crashes. Otherwise, it may violate the specification of atomic broadcast. Consider for instance a message m abcast by process p. If p crashes while sending a copy of m to all processes, m may be delivered at some processes but not at others. This violates reliable broadcast's specification (see [13]). Moreover, in this example, it may also lead to a violation of atomic broadcast's specification. To avoid this in our implementation (and thus ensure correctness), if a process q does not receive messages during a period of t seconds (with t sufficiently big), q starts a consensus even if no message arrives.

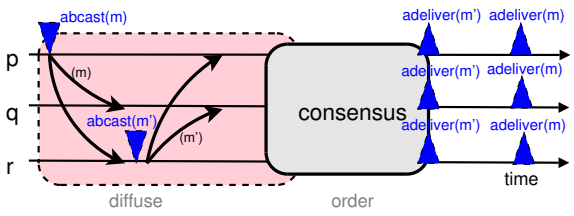

Figure 4. Example run of atomic broadcast by reduction to consensus.

4 Monolithic Implementation

In the previous sections, we have presented the algorithms (and optimizations) as they are implemented in the modular atomic broadcast stack. When we implement these algorithms as a single module in a monolithic stack, further (algorithmic) optimizations are possible. In this section, we present the optimizations that were carried out in the monolithic stack (see Fig. 1, right). Again, our optimizations focus on good runs but ensure correctness in all runs.

For each of these optimizations we explain (1) what changes are made compared to the modular version of atomic broadcast (see Section 3), (2) why these changes are possible, and (3) what (approximate) improvement in performance can be expected from these changes.

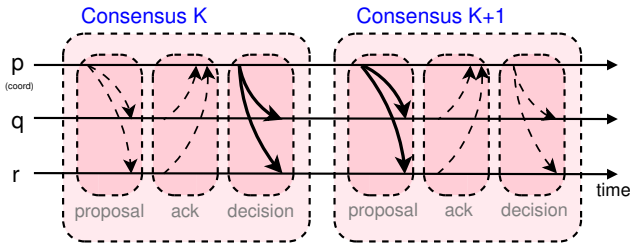

Figure 5. Consecutive consensus executions in the modular implementation of atomic broadcast.

4.1 Combining the Next Proposal with the Current Decision

In the modular implementation of atomic broadcast (see Fig. 4), atomic broadcast runs a sequence of consecutive consensus instances to order the set of undelivered messages. Due to the modular design, all consensus instances are black boxes from the point of view of atomic broadcast and are considered to be totally independent from each other. Thus, we cannot take advantage of the fact that the coordinator that sends a decision in consensus instance k is the same coordinator that sends a proposal in consensus instance $k+1$. Figure 5 shows this. Note that in normal executions, process p does not necessarily wait until processes q and r decide to start another consensus. In other words, process p may send its proposal for consensus instance $k+1$ just after having sent the decision of consensus instance k.

In the monolithic implementation the successive consensus instances are run within the atomic broadcast module. If consensus instance k decides in the first round (which is the case in good runs), then the coordinator of consensus instances k and $k+1$ (in its first round) are the same process. In this case, the decision of consensus k and the proposal of consensus $k+1$ are sent together as a single message (denoted "proposal k + decision $k-1$" in Figure 6).

This first optimization in the monolithic atomic broadcast stack allows a better use of network resources: instead of sending one small message (tag DECISION) followed by a larger message (the proposal of consensus $k+1$) the small message is piggybacked on the larger one.

4.2 Piggybacking Messages Abcast on *ack* Messages

In the modular implementation of atomic broadcast, a process abcasting a message m starts by sending m to all other processes. Whenever this message is received, it is added to the set of proposed messages for the next consensus instance. In good runs, this is inefficient for the following reason: every process delivers m, but only the coordinator of the next consensus execution actually needs m (in or-

0-7695-2855-4/07 $20.00

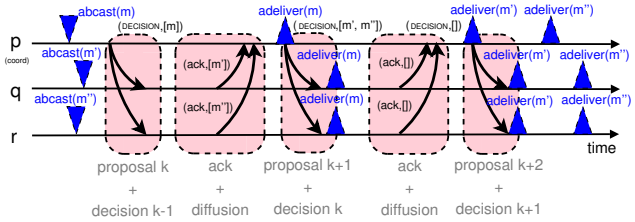

Figure 6. Consecutive consensus executions in the monolithic implementation of atomic broadcast.

der to propose m for consensus). This can not be optimized in a modular stack, since the atomic broadcast module cannot access information that is specific to the consensus module (such as the identity of the coordinator). Furthermore, to preserve modularity, atomic broadcast can not disseminate messages abcast by the application within consensus messages. This is shown in Figure 7: messages m and m' are first sent (in the diffuse step), then consensus is executed (in the order step).

A more efficient solution, which can only be implemented in the monolithic stack, is to combine *ack* messages with the sending of messages m and m' (see solid arrows in Fig. 7). This is done as follows. The sender of m *directly* sends m to the (initial) coordinator of the next consensus execution. Furthermore, instead of sending m as a standalone message to the coordinator, it can be piggybacked on the *ack* message of the consensus algorithm (denoted "ack + diffusion" in Figure 6). If the coordinator changes (i.e. if a suspicion occurs), message m is again piggybacked on the estimate sent to the new coordinator.

The gain of this optimization is twofold. Firstly, it reduces network congestion by avoiding an unnecessary diffusion of abcast messages to all processes: messages are only sent to the coordinator. Secondly, similarly to the first optimization presented above, it allows a more efficient use of network resources thanks to the aggregation of small messages with bigger ones.

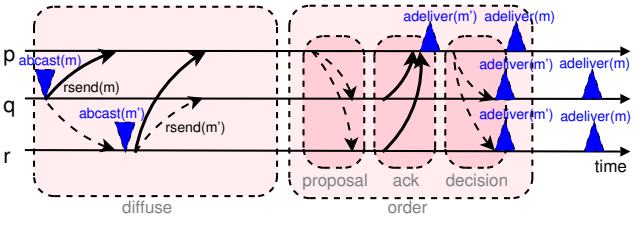

Figure 7. Diffusion of two messages m and m', followed by their ordering. The diffusion and ordering steps cannot be merged in the modular implementation of atomic broadcast.

4.3 Reducing the Message Complexity of Reliable Broadcast

Consensus decisions have to be reliably broadcast to all processes. In the modular implementation, the reliable broadcast algorithm requires $(n-1)\cdot(\lfloor\frac{n+1}{2}\rfloor)$ messages to be sent on the network for each reliable broadcast to n processes.

In the monolithic implementation, the cost of the decision diffusion is reduced to n messages: the decision is simply sent to all processes without any additional retransmissions (in good runs). The reduction relies on the knowledge that the successive consensus instances are executed on the same set Π of processes (and thus, messages in consensus $k+1$ can serve as acknowledgments for messages sent in consensus k). With this knowledge, the decision of consensus execution k is acknowledged by the messages sent by non-coordinators to the coordinator in consensus execution $k+1$.

Again, this optimization reduces the network congestion since it considerably reduces from $(n-1)\cdot(\lfloor\frac{n+1}{2}\rfloor)$ to n the number of messages sent by reliable broadcast.

5 Performance Evaluation

We now evaluate and compare the performance of our two (optimized) implementations of atomic broadcast. We specifically focus on the case of a system with three and seven processes, supporting one, respectively three, failures. This system size might seem small. However, atomic broadcast is usually used for relatively small degrees of replication. If a a larger degree of replication is needed, then alternatives that provide weaker consistency should be considered [1].

The section starts by presenting the parameters considered. An analytical evaluation of the two implementations is then presented, followed by the experimental evaluation of these implementations.

5.1 Metrics, Workload, Faultload

The following paragraphs describe the benchmarks (i.e. the performance metrics and the workloads) that were used to evaluate the performance of the atomic broadcast algorithms.

Performance Metrics. We use two performance metrics to evaluate the algorithms: *early latency* and *throughput*. For a single abcast message, the early latency L is defined as follows. Let t_0 be the time at which the $abcast(m)$ event completes and let t_i be the time at which $adeliver(m)$ occurs on process p_i, with $i \in 1,\ldots,n$. The early latency L is then defined as $L \stackrel{def}{=} (\min_i t_i) - t_0$.

The throughput T is defined as follows. Let r_i be the rate at which $adeliver$ events occur on a process p_i, with $i \in 1, \ldots, n$. The throughput T is then defined as $T \stackrel{def}{=} \frac{1}{n} \sum_{i=1}^{n} r_i$ and is expressed in messages per second (or msgs/s).

In our performance evaluation, the mean for L and T is computed over many messages and for several executions. For all results, we show 95% confidence intervals.

Workloads and Faultload. The early latency L and the throughput T are measured for a certain workload, which specifies how many *abcast* events are generated per time unit. We chose a simple symmetric workload where all processes abcast messages of a fixed size s at a constant rate r (with s and r varying from experiment to experiment). The global rate of atomic broadcast events is called the *offered load* $T_{offered}$, which is expressed in messages per second. We then evaluate the dependency between, on one hand, the early latency L and the throughput T and, on the other hand, the offered load $T_{offered}$ and the size of the messages.

Furthermore, both implementations of the atomic broadcast protocol use the same flow-control mechanism that blocks further *abcast* events when necessary. More precisely, the flow-control mechanism ensures that, on average, $M = 4$ messages are ordered per consensus execution. This value of M optimizes performance of both modular and monolithic stacks. We ensure that the system stays in a stationary state by verifying that the latencies of all processes stabilize over time.

Finally, we only evaluate the performance of the algorithms in good runs, i.e. without any process failures or wrong suspicions. The latency and throughput of the implementations is measured once the system has reached a stationary state (at a sufficiently long time after the startup). The parameters that influence the latency and the throughput are n (the number of processes), the implementation (modular or monolithic) the offered load $T_{offered}$ and the size of the messages that are abcast.

5.2 Analytical Evaluation

As shown in Section 3, the Chandra-Toueg atomic broadcast algorithm reduces to a sequence of consensus executions. We assume a workload high enough so that consensus execution $k+1$ starts directly after execution k.[4] This condition is met if the offered load $T_{offered}$ is greater than the number of consensus executions that the system can execute per second (i.e., if d is the average duration of a consensus execution, we have $T > d^{-1}$).

We now analyze two aspects of the performance of the two implementations: (1) the number of messages that are sent and (2) the total amount of data that needs to be sent to solve atomic broadcast.

5.2.1 Number of Sent Messages

In both the modular and monolithic implementations of atomic broadcast, sets of unordered abcast messages are ordered in consensus executions. We assume that, on average, M messages are ordered per consensus execution. In the experimental evaluation, this is ensured by our flow-control mechanism. We now derive the number of messages that need to be sent in both stacks in order to adeliver these M messages.

Modular Implementation. In the modular implementation of atomic broadcast, the M unordered messages are first sent to all processes in the system, generating $M \cdot (n-1)$ messages on the network. These messages are then received by the coordinator of the consensus algorithm that sends a proposal to all processes ($n-1$ messages). All processes reply by sending an *ack* message to the coordinator ($n-1$ messages), which then reliably broadcasts the decision to all processes (which necessitates an additional $(n-1) \cdot \lfloor \frac{n+1}{2} \rfloor$ messages).

To adeliver the M abcast messages, the modular implementation thus needs to send $(n-1)(M+2+\lfloor \frac{n+1}{2} \rfloor)$ messages.

Monolithic Implementation. In the monolithic implementation of atomic broadcast, the M unordered messages are not immediately sent to all processes. Instead, they are piggybacked on the *ack* messages of the previous consensus execution. The coordinator starts the consensus execution by sending both the decision of the previous consensus and a new proposal in the same message. This message is sent to all processes ($n-1$ messages). The other processes then reply by sending an *ack* message to the coordinator ($n-1$ messages).

To adeliver the M abcast messages, the monolithic implementation thus only needs to send $2 \cdot (n-1)$ messages.

In the case of a system of $n = 3$ processes for example, with an average of $M = 4$ messages ordered per consensus execution[5], this means that the monolithic implementation needs 4 messages to order these 4 abcast messages (assuming of course that a previous consensus execution allows some piggybacking of messages). In the case of the modular stack, 16 messages are needed for the same result.

[4] Otherwise, there is no point in optimizing the algorithms.

[5] This value of M corresponds to the one that we chose for our experimental evaluation.

5.2.2 Total Amount of Sent Data

We assume that abcast messages all have a size of l bytes. We further assume that messages sent by the algorithm that have a constant size (e.g. *ack* messages and tag DECISION in the modular implementation) only represent a negligible part of the sent data. As above, we analyze how much data is sent on average per consensus execution (i.e., to adeliver M abcast messages).

Modular Implementation. In the modular implementation, abcast messages are sent to all other processes. The M messages of size l are thus sent to $n-1$ processes. The coordinator then adds these messages to a consensus proposal (sent to the $n-1$ non-coordinator processes) which thus has a size of $M \cdot l$ on average. The total amount of data exchanged per consensus in the modular stack is then $Data_{mod} = 2(n-1)M \cdot l$ bytes.

Monolithic Implementation. In the monolithic implementation, the processes do not diffuse their abcast message to everyone and instead only send them to the coordinator (by piggybacking them on *ack* messages). On average, $\frac{M}{n}$ abcast messages of size l are piggybacked by each one of the $n-1$ non-coordinator processes during a consensus execution. The coordinator then creates a proposal with the M messages ($\frac{M}{n}$ messages abcast by itself and $(n-1)\frac{M}{n}$ abcast by the other processes) of size l that is sent to the $n-1$ other processes. The total amount of data sent per consensus execution is thus on average $Data_{mono} = (n-1)(1+\frac{1}{n})M \cdot l$ bytes.

The overhead of the modular implementation with respect to the monolithic implementation is therefore

$$overhead = \frac{Data_{mod} - Data_{mono}}{Data_{mono}} = \frac{n-1}{n+1}$$

In a system with $n = 3$ processes, the modular implementation needs to send 50% more data than the monolithic one. In the case of $n = 7$, the overhead reaches 75%.

5.3 Experimental Evaluation

The paragraphs above presented an analytical evaluation of the two atomic broadcast implementations from the perspective of two performance aspects. These two aspects are however not sufficient to completely characterize the performance cost of the modular implementation versus the monolithic one. Indeed, the analysis above focuses on aspects related to the network communication of the two implementations, whereas processing times for example are not at all taken into account. The experimental evaluation of the two stacks fills this gap.

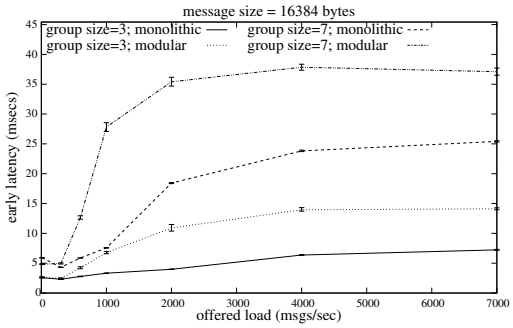

Figure 8. Early latency vs. offered load for abcast messages of size 16384 bytes.

The following paragraph presents the system setup used in the experiments. Then, a performance comparison is presented between the modular and monolithic stacks.

5.3.1 System Setup

The benchmarks were run on a cluster of machines running SuSE Linux (kernel 2.6.11). Each machine has a Pentium 4 processor at 3.2 GHz and 1 GB of RAM. The machines are interconnected by Gigabit Ethernet (which is exclusively used by the cluster machines) and run Sun's 1.5.0 Java Virtual Machine. The machines were dedicated to the performance benchmarks and had no other load on them.

The atomic broadcast algorithm was implemented (twice) in Fortika ver. 0.4[6] [18, 19]. Fortika is a group communication toolkit with various well-known off-the-shelf protocol modules. These protocol modules can then be composed using different protocol frameworks. The current experiments were run with the Cactus protocol framework [4, 24].

5.3.2 Performance Results

Latency of Atomic Broadcast. Figure 8 shows the evolution of the early latency (vertical axis) of atomic broadcast using the two implementations as the offered load (horizontal axis) increases. Results are shown for a system size of $n = 3$ (two bottom curves) and $n = 7$ (two top curves), with abcast messages of size 16384 bytes. Note that changing the size of messages does not significantly affect the results.

The latency of both implementations is relatively close for small offered loads. As soon as the offered load increases, however, the monolithic implementation achieves latencies that are between 30% ($n = 7$) and 50% ($n = 3$) lower than the modular implementation. Note that the latency of the two implementations remains relatively con-

[6]The current version of Fortika uses TCP connections rather than IP multicast facilities.

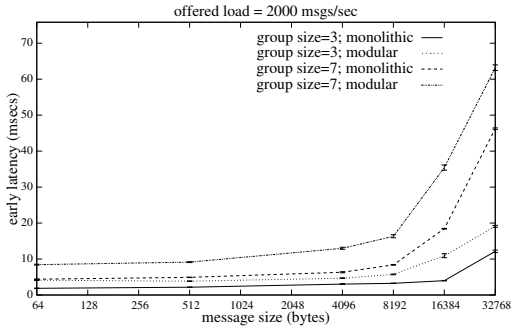

Figure 9. Early latency vs. message size for an offered load of 2000 msgs/s.

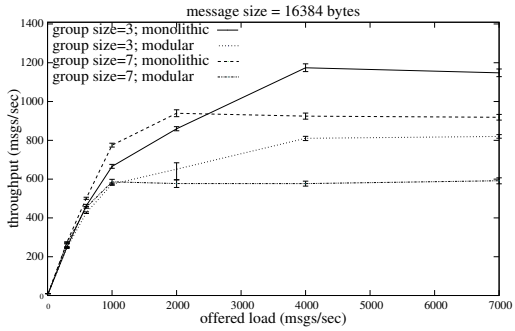

Figure 10. Throughput vs. offered load for abcast messages of size 16384 bytes.

stant above a certain offered load. This is due to the flow-control mechanism that is present in both stacks: as the offered load increases, more and more abcast messages are blocked so that the network load remains more or less constant.

Figure 9 shows how the early latency of the two implementations is affected by the size of the messages that are abcast. The graph shows the early latency in a system with $n = 3$ (two bottom curves) and $n = 7$ (two top curves) processes. The offered load is fixed to 2000 msgs/s. Results are similar with other values of offered load (except with too small values where no significant differences can be observed).

Once again, the monolithic implementation achieves latencies about 50% lower than the modular implementation when the size of the messages is small (up to 4096 bytes when $n = 7$ and 8192 bytes when $n = 3$). When the size of the messages increases, the early latency also increases: here, the total amount of data that needs to be exchanged influences the latency, whereas previously the latency was determined mostly by the number of messages sent on the network (these messages all require a certain amount of processing, independently of their small size). Finally, with the largest messages considered, the monolithic implementation achieves a latency that is 25% ($n = 7$) or 35% ($n = 3$) smaller than the modular implementation.

Throughput of Atomic Broadcast. We now examine what throughput is reached by the modular and monolithic implementations of atomic broadcast. Figure 10 shows the relationship between the throughput of atomic broadcast (on the vertical axis) and the offered load (on the horizontal axis) when the size of the atomic broadcast messages is fixed at 16384 bytes. When the offered load is small (less than 500 msgs/s), the throughput is equal to the offered load. As the offered load increases, the flow-control mechanism limits the throughput that can be achieved (as in the early latency above, the throughput reaches a plateau as the offered load increases). Furthermore, for a high offered

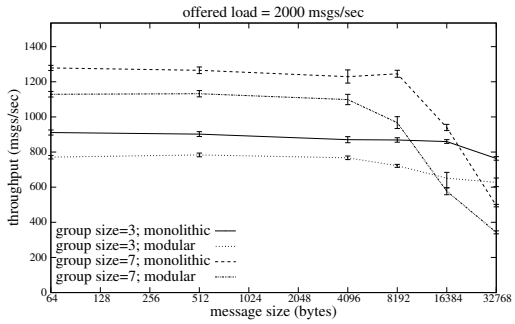

Figure 11. Throughput vs. message size for an offered load of 2000 msgs/s.

load, the monolithic implementation sustains a throughput that is 25% ($n = 7$) to 30% ($n = 3$) higher than the modular implementation. For a low offered load, the difference between both stacks is almost negligible.

Figure 11 presents the throughput of both implementations as a function of the size of the messages that are abcast. The offered load is fixed at 2000 msgs/s. When the size of the messages is small, the monolithic implementation achieves between 10% and 15% higher throughputs than the modular one (and the throughput remains constant up to messages of size 4096 for $n = 7$ and 16384 for $n = 3$). Surprisingly, the throughput is higher when $n = 7$ processes participate in the system than when $n = 3$. This is once again due to the flow-control mechanism: each process is allowed to have a certain backlog (i.e. abcast messages that have not been delivered yet). Hence, when the number of processes grows, a larger number of abcast messages that have not been adelivered are allowed to circulate.

Finally, as the message size increases, the throughput of the system with $n = 7$ processes degrades faster than in the case of $n = 3$. This is due to the consensus proposal (which contains large messages) that needs to be sent to all processes in the system. As both the message size of and the number of processes increase, sending these large proposals results in an overall lower throughput (in msgs/s).

Discussion. From the results above, we see that the difference in performance between a modular and a monolithic implementation of the same distributed protocol is significant: the difference in latency is up to 50%, while the difference in throughput varies between 10% and 25%. This is the cost that a user must expect to pay when choosing between a modular system that is easier to maintain and update and a monolithic system that has better performance characteristics.

Furthermore, it is interesting to note that the experimental results do not always match the analysis in Section 5.2. These two results are, however, complementary. As explained earlier, the analytical evaluation of the two implementations focuses solely on the messages exchanged by the algorithm. Processing costs and resource contention, for example, are not at all considered in the analysis. On the other hand, in evaluating throughput and message latency, experimental results are strongly influenced by such elements (but do not consider explicitly number of messages exchanged). For instance, 99% of CPU resources were used with an offered load bigger than 500 msgs/s. The discrepancy between the analytical and experimental evaluations of the stacks stem from these elements (that are difficult to estimate a priori).

6 Related Work

A number of group communication toolkits have implemented atomic broadcast during the last two decades. While early implementations (Isis [5, 6], Phoenix [17] and Totem [2], among others) were designed with a monolithic architecture, more recent systems (Horus [27], Ensemble [14], Transis [11], JavaGroups [3], Eden [15], and Fortika[18, 19][7]) present a modular design. A comparison, from the architectural point of view, of most of these group communication toolkits can be found in [20]. However, the issue of performance overhead induced by modularity (i.e. comparing performance of a modular and a monolithic stack based on the same architecture) has not been covered extensively. In Ensemble, the performance was improved through several techniques [26, 14]: optimizing the interfacing code, improving the format of headers from different modules and compressing them, extracting and inlining frequently executed functions (from many modules), etc. Appia, a system inspired by Ensemble, included and furthered these techniques [21]. While these techniques significantly improved the performance of these systems (e.g., in [14], they reduce by approximately 20 the time of processing), they are rather general lower-level solutions. Their aim is not at the algorithmic level: the algorithms stayed the same after the optimizations. On the other hand, our algorithmic improvements can not be applied to Ensemble or Appia, where atomic broadcast is not solved by reduction to consensus, but rather relies on group membership in order to avoid blocking. In [12], the authors propose to extend the specification of consensus. The new specification allows the consensus layer to share some state with the above layers (e.g. atomic broadcast) in order to reduce the amount of data sent over the network. This technique improves significantly performance (reduction by 4 of the message latency). However, this result is not comparable to current result due to significant differences in the system setup. Note that the Eden group communication toolkit [15] proposes a very similar technique.

In a more general context, there is more extensive work on protocol layer optimization. For instance, the influential x-Kernel modular system was improved with the help of various techniques like protocol multiplexing [23]. Standard compilation techniques can be combined with annotations in the code to optimize the most frequently executed functions [22]. This approach is somewhat similar to the work done in Ensemble, but for more more basic stacks like TCP/IP. Another technique to improve performance across a protocol stack is Application Level Framing [9, 10]. The intuition here is that all protocols should know the typical size of application messages, so that they are not unnecessarily fragmented on their way down the stack. Again, in all these techniques, protocols are treated as black boxes: the optimizations did not involve any modification in the protocol logic. Hence, most of these techniques can easily be combined with the ones proposed in this paper.

Modularity is a necessary property to achieve good performance in parallel computing and concurrent programming [16]. However, this is not applicable to our work, since very few tasks can be parallelized in atomic broadcast: only message diffusion and ordering can be executed concurrently.

7 Conclusion

The paper presented two versions (monolithic and modular) of a fairly complex protocol: atomic broadcast. We showed that a monolithic stack allows several algorithmic optimizations. This is principally due to (1) the fact that consensus instances are not considered independently, and (2) the possibility for different modules to share their state. We then analytically and experimentally quantified the gain obtained thanks to these optimizations. Our analytical evaluation concluded that a monolithic implementation significantly reduces the number of messages sent over the network. On the other hand, our experimental evaluation revealed an overhead incurred by the modular version that reaches 50% in the worst workload conditions.

[7] Actually, Fortika provides both modular and monolithic implementations of atomic broadcast.

In summary, if we are to implement atomic broadcast, it is commonly agreed that a modular design is the most sensible approach. In this paper, we have shown that we cannot be so sure of this (apparently undisputed) choice, if we care about our system's performance.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and helpful suggestions.

References

[1] L. Alvisi and K. Marzullo. Waft: Support for fault-tolerance in wide-area object oriented systems. In *Proc. of the 2nd Information Survivability Workshop – ISW '98*, pages 5–10. IEEE Computer Society Press, October 1998.

[2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, Nov. 1995.

[3] B. Ban. *JavaGroups 2.0 User's Guide*, Nov 2002.

[4] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, Nov. 1998.

[5] K. P. Birman. The process group approach to reliable distributed computing. *Comm. ACM*, 36(12):36–53, Dec. 1993.

[6] K. P. Birman and T. A. Joseph. Reliable communication in presence of failures. *ACM Trans. on Computer Systems*, 5(1):47–76, Feb. 1987.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, Mar. 1996.

[8] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, May 2001.

[9] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*, pages 200–208, New York, NY, USA, 1990. ACM Press.

[10] J. Crowcroft, J. Wakeman, Z. Wang, and D. Sirovica. Is Layering Harmful? IEEE Network 6(1992) 1 pp. 20-24. *IEEE Network 6(1992) 1 pp. 20-24*, 1992.

[11] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Comm. ACM*, 39(4):64–70, Apr. 1996.

[12] R. Ekwall and A. Schiper. Solving atomic broadcast with indirect consensus. In *2006 IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, 2006.

[13] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[14] M. Hayden. The Ensemble system. Technical Report TR98-1662, Dept. of Computer Science, Cornell University, Jan. 8, 1998.

[15] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS)*, pages 56–67, Lausanne, Switzerland, Oct. 1999.

[16] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[17] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 1996.

[18] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia *vs.* cactus: Comparing protocol composition frameworks. In *Proc. of 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, Oct. 2003.

[19] S. Mena, O. Rütti, and A. Schiper. *Fortika: Robust Group Communication*. EPFL, Laboratoire de Systèmes Répartis, may 2006.

[20] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of Conference on Middleware*, Rio de Janeiro, Brasil, June 2003.

[21] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st Int'l Conf. on Distributed Computing Systems (ICDCS' 01)*, pages 707–710, Washington - Brussels - Tokyo, Apr.16–19 2001.

[22] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 1996. ACM Press.

[23] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abbott. Rpc in the x-kernel: evaluating new design techniques. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 91–101, New York, NY, USA, 1989. ACM Press.

[24] The University of Arizona, Computer Science Department. *The Cactus Project*. Available electronically at `http://www.cs.arizona.edu/cactus/`.

[25] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 2003. Number 2824.

[26] R. van Renesse. Masking the overhead of protocol layering. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 96–104, New York, NY, USA, 1996. ACM Press.

[27] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, Apr. 1996.