Flexible Protocol Composition in Bast *

Benoît Garbinato Rachid Guerraoui

Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne, Suisse

Abstract

This paper presents BAST, *an object-oriented library of reliable distributed protocols. We show how* BAST *can be used to build fault-tolerant distributed applications, and how new protocols can be added to it. We discuss some distributed protocol design issues and the way these issues are circumvented in* BAST. *We briefly describe our Smalltalk and Java implementations of* BAST, *together with some performance results.*

1 Introduction

Reliable distributed applications are challenging to build, because programmers have to deal with many complex issues, e.g., reliable communication, failure detection, replication management, transaction management, etc. Each of these issues actually corresponds to a distributed problem to solve, and programmers have to face a distributed problems "jungle" (Figure 1) to choose the right protocol for the right problem, or to compose existing protocols and make them work together.

BAST[1] is an extensible object-oriented library of reliable distributed protocols. BAST helps managing the complexity of programming reliable distributed applications, by allowing to structure fault-tolerant distributed applications according to their needs in protocols, and by simplifying the composition of complex protocols in a flexible manner [5]. Protocol composition in BAST makes it possible to easily build new protocols from existing ones. An *application programmer* can directly *use* built-in BAST protocol classes, and a *protocol programmer* can *extend* BAST with new protocols.

In contrast to systems like ARJUNA [12], MELD [14] and KAROS [8] on one hand, or HORUS [18] and CONSUL [11] on the other hand, BAST is neither a transaction-oriented nor a group-oriented library. BAST provides very abstract building blocks such as a generic *agreement* and a *failure detection* protocol. More specific protocols such as *transaction atomic commitment* [1] or *group membership* [2] are viewed as higher level protocols that can be used or not according to the application's semantics. As a consequence, BAST does not impose any reliable distributed programming paradigm. One can for instance develop a transaction-oriented or a group-oriented distributed application, or even mix, within the same application, both transactions and group communications [9].

Several attempts to structure distributed protocol interactions adopted a strictly vertically layered approach (e.g., Horus). Although this *black box* approach provides a nice transparency feature for the application programmer, it reduces the possibility of protocol extension (as noticed in [11] for instance). Among the *white box* approaches that we know of, several rely on inheritance to derive new protocols from existing ones (e.g., ARJUNA [12] and ACE [17]). As we point out later in the paper (Section 3), single inheritance alone does not lead to optimal code reuse, and multiple inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*, or to dealing with concurrent executions of the same protocol.

In BAST, distributed algorithms are *objectified*, i.e., considered as first-class objects. Only the *encapsulation* benefit of the layered approach was retained, while its strictly vertical nature was discarded. Furthermore, inheritance is still used as a convenient code reuse mechanism at design time, but runtime problems are handled naturally when each protocol execution is materialised as a separate algorithm object. Although applied in different contexts, our approach is similar to those followed in the design of the CONDUIT+ framework for network protocols [10], and the x-KERNEL [13] library of communication protocols.

Our first implementation of BAST was written in Smalltalk. It has been used for teaching reliable distributed systems, as well as for prototyping new fault-tolerant distributed protocols. BAST was later ported

*Partially supported by OFES under contract number 95.0830, as part of the ESPRIT BROADCAST-WG (number 22455).

[1] In the Egyptian mythology, *Bast* is a cat-goddess.

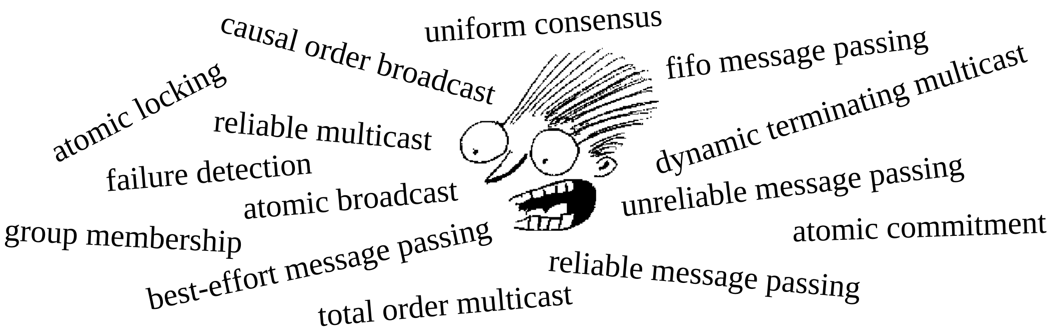


Figure 1: The Distributed Problems "Jungle"

to Java, and a Java prototype is now fully operational.

Section 2 presents how programmers can use BAST protocol classes to build fault-tolerant distributed applications. Section 3 discusses the general problem of distributed protocol interaction, and presents the way this problem is dealt with in BAST. Section 4 gives an overview of how new protocol classes can be created by composing existing ones. Section 5 discusses implementation issues, and presents some performance measures. Section 6 points out some concluding remarks.

2 Using BAST

Building fault-tolerant applications is normally a hard task which requires highly skilled *protocol programmers*, because reliable distributed protocols can be very complex to understand and to implement. Ready-to-use distributed protocols make this task less error prone and more accessible to *application programmers*. For example, an atomic commitment protocol is essential to any application involving distributed transactions, while an atomic broadcast protocol is the cornerstone of applications based on active replication. BAST provides such ready-to-use protocols in the form of *protocol classes*.

2.1 Protocol Classes

In BAST, a protocol class π`Object` implements the behaviour of remotly addressable objects that are capable of executing distributed protocol π, i.e., of solving distributed problem π. Instances of protocol classes are designated as *protocol objects*. We use π`Object` to refer to any protocol class, and `a`π`Object` to designate some protocol object of class π`Object`. Distinct protocol objects reside *a priori* on different nodes, and faulty nodes are supposed to fail independently by crashing. We do not make any assumption on the synchrony of the system, but we use the notion of failure detector instead[2].

Figure 2 presents some ready-to-use protocol classes proposed in BAST. This class diagram uses the standard OMT (Object Modelling Technique) notation [16]; in Figure 2, abstract operations are all callback operations. The root of this class hierarchy is the abstract superclass `Protobject`. Each π`Object` class provides a set of operations that interface protocol π: these operations act as entry points for applications using protocol π. For example, class `RMPObject`, which implements reliable point-to-point communications, provides operations `rSend()` and `rDeliver()` that allow to reliably send, respectively receive, a message. It is important to notice that, in BAST, even *simple* distributed protocols such unreliable communication are implemented using protocol objects. We believe that the generality of this definition helps the uniform structuring of the library.

2.2 Genericity of Protocol Classes

In BAST, distributed protocols are *generic* in the sense that: (1) the type of arguments passed to any protocol operation is not restricted; e.g., operation `rSend()` can be used to send *any* object across the network, including lists and trees; (2) callback operations, such as `rDeliver()`, are supposed to be redefined by each application in order to meet its needs; callbacks are said to be *triggered by the protocol*.

Protocol classes are organised into a single inheritance hierarchy: each protocol class implements only one protocol, but instances of some π`Object` class can execute any protocol inherited from π`Object`'s superclasses (see Figure 2). Protocol objects are capable of running several executions of identical and/or distinct protocols concurrently (we will come back to this issue in Section 3).

[2]Failure detectors enable to hide the synchrony of distributed systems and are abstractly defined through reliability properties [3].

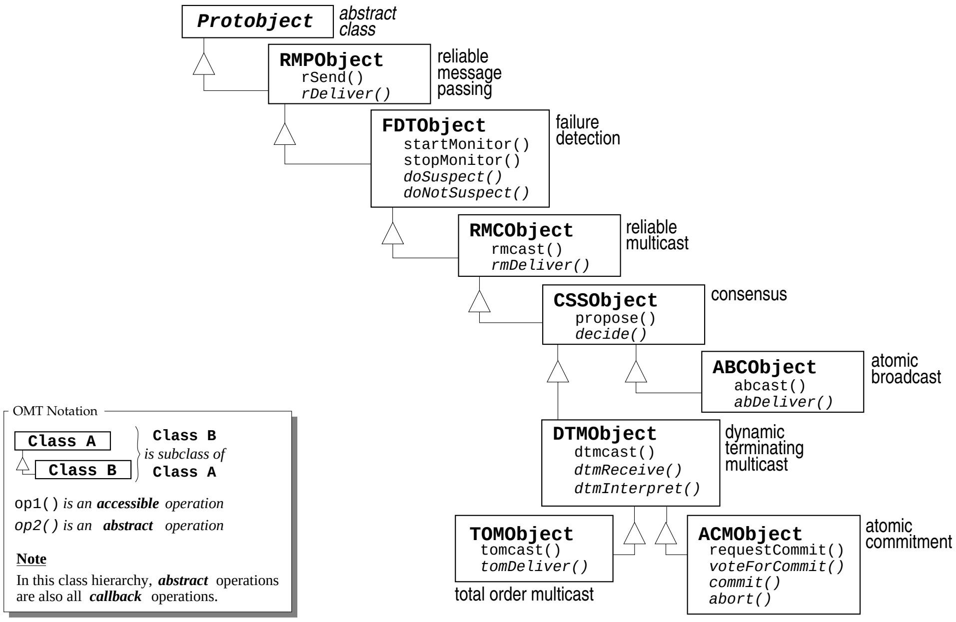


Figure 2: Some Protocol Classes in BAST

3 Protocol Dependencies in BAST

Since relationships between distributed protocols can be quite complex, a first step towards protocol code reuse is to structure this complexity. In the following, we discuss some design alternatives in assembling distributed protocol layers, and we describe the solution we adopted in BAST.

3.1 Inheritance as Composition Tool

Fault-tolerant distributed protocols are often challenging to build, because they involve many other underlying protocols. This can result in complex dependency relationships. The atomic broadcast protocol implemented in BAST, for instance, is built on top of consensus and reliable multicast communication. Consensus is itself based on failure detection, on reliable point-to-point communication, and on reliable multicast communication; in turn, the latter also relies on a reliable point-to-point communication.

As presented in Section 2, inheritance is an appropriate mechanism to build fault-tolerant distributed applications from ready-to-use components: by subclassing appropriate protocol classes and by implementing their callback operations according to the desired semantics, application programmers have the ability to tailor protocols to their needs. However, inheritance alone is not sufficient as far as protocol composition goes, because it does not offer enough flexibility. For example, inheritance does not allow to easily implement a new algorithm for some existing protocol and to use it in whatever protocol class we want.

We illustrate this situation in Figure 3: given the pictured class hierarchy, suppose some protocol programmer wants to change the reliable multicast algorithm on which both the causal and the atomic broadcast protocols rely (classes `CBCObject` and `ABCObject` respectively), while leaving it unchanged in all other classes (classes `CSSObject` and `DTMObject` in our case). With inheritance alone as code reuse mechanism, one has to implement the new reliable multicast algorithm in both `CBCObject` and `ABCObject` classes, hampering optimal code reuse. Furthermore, inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*.

3.2 Mixins as Design Alternative

Assembling the various protocol layers through multiple inheritance can be viewed as an appealing alternative: each π`Object` class would implement only protocol π, while accessing all required underlying protocols through unimplemented operations. The latter would then be provided by other protocol classes through multiple inheritance. With such a design, protocol classes are all *abstract* and we usually talk about *mixin classes* or simply *mixins*.

There are several drawbacks with this approach, however. First, protocol classes are not more ready-to-use components: before being able to actually create

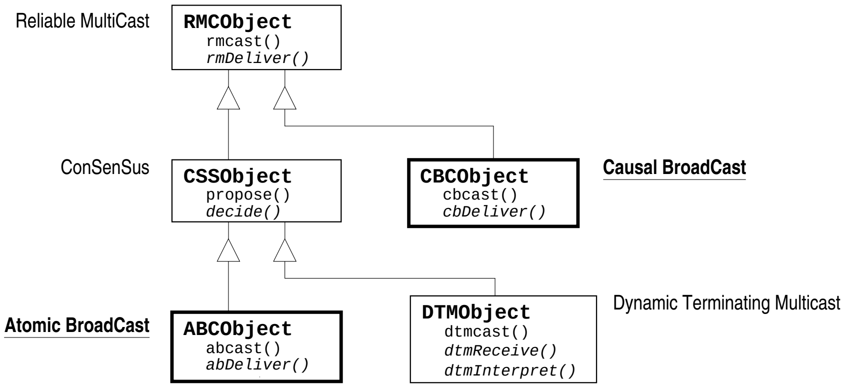


Figure 3: Problems with Inheritance

a protocol object, one first has to build a new class deriving from all the necessary protocol classes. Application programmers no more can ignore protocol dependencies. Then, because protocol layers are assembled through subclassing, it is very difficult to compose protocols at runtime. While fairly complex yet possible in Smalltalk, such run-time protocol composition is impossible in Java (classes can only be created at compile-time in this language). Finally, concurrent protocol executions within the same protocol object have still to be managed, whereas this problem is handled naturally when each protocol execution is materialised as a separate algorithm object, as we shall see now.

3.3 Objectifying Distributed Algorithms

These limitations lead us to *objectify* distributed algorithms, i.e., to make separate first-class objects out of them, as presented in Section 4. As a result, each protocol object maintains a runtime table of all private algorithm objects concurrently in execution for it; messages are tagged in order to identify in which execution of what protocol they have to be delivered. Protocol dependencies are thus managed during the design and implementation phases (between protocol classes), and also at runtime (between protocol objects). This is partly due to the fact that protocol objects can execute more than one protocol at the same time.

Figure 4 (a) presents an instance of class `ABCObject` during its execution. Class `ABCObject` defines two entry operations to the atomic broadcast protocol: `abcast()` and `abDeliver()`. Besides those operations, protocol object `anABCObject` is also capable of concurrently executing any protocol inherited by its class, e.g., consensus and reliable multicast communications (see class hierarchy in Figure 2). As shown in the figure, `anABCObject` is simultaneously running five different protocols on behalf of the application layer, while performing low-level calls to the transport layer for this job.

4 Composing Protocols in Bast

In Bast, protocol dependencies are modelled as in the *OSI* (Open Systems Interconnection) stack model [15].

4.1 Protocol Layers Interactions

As depicted in Figure 4 (b) (class `ABCObject`), each protocol execution is supported by an independent protocol stack. Hence, each time the application layer invokes one of its operations, a new protocol stack is created. Furthermore, every layer of that stack is in charge of only one protocol supported by class `ABCObject`. In the context of `ABCObject`, protocol composition means here to assemble various layers, each one running a protocol algorithm necessary to the execution of the atomic broadcast protocol. This is where protocol composition actually occurs, as we discuss below.

4.2 Distributed Algorithms as Objects

In order to separate protocol layers from their implementations, distributed algorithms are manipulated as first-class objects. Consequently, each π`Object` protocol class is independent of the algorithm supporting protocol π, which makes protocol composition fully flexible. Elsewhere [6], we show how objectifying protocol algorithms comes down to recursively apply the *Strategy design pattern*[3] introduced by Gamma and al. [4].

Distributed algorithm objects of class π`Algo` are privately used by the corresponding π`Object` protocol objects and hold the actual implementation of distributed algorithm π. An algorithm object always executes *within* a protocol object: as a result, application

[3]Design patterns capture recurring object-oriented solutions and help to reuse expertise in software design.

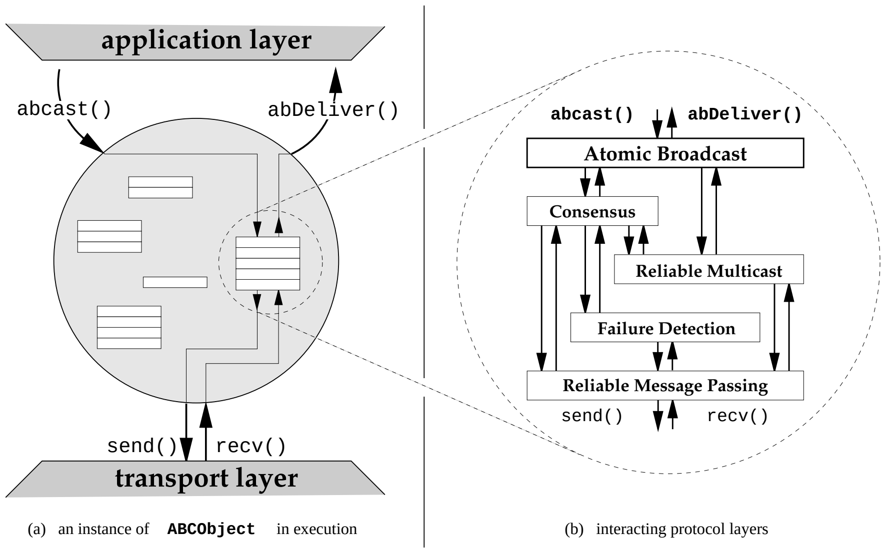


Figure 4: Protocol Objects in BAST

programmers only deal with protocol objects and know nothing about algorithm objects.

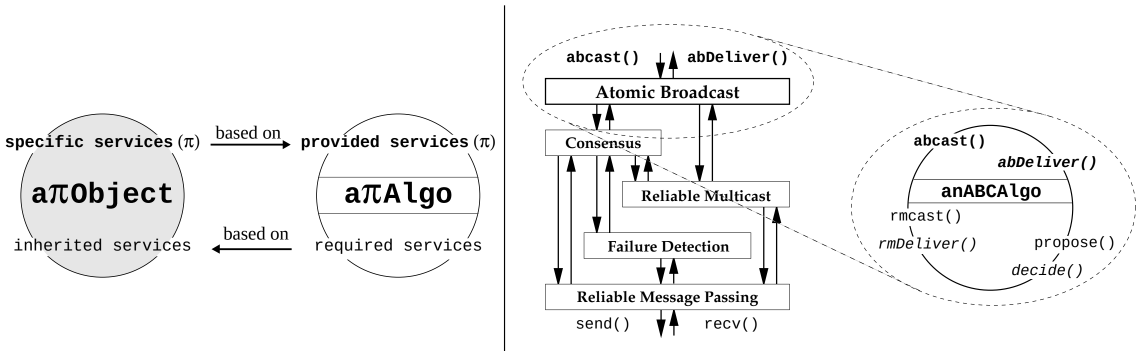
Protocol algorithm classes all derive from abstract class `ProtoAlgo`, which defines a completely separate inheritance hierarchy from the protocol objects' one. Figure 5 (a) illustrates the relationship between aπ`Object`, some protocol object, and aπ`Algo`, one of its private algorithm objects. In this Figure, we group operations offered by protocol object aπ`Object` in two categories: those it inherits from its superclasses and those that are specific to protocol π. For the latter operations, their actual algorithm is implemented in object aπ`Algo`'s code. In turn, all other protocols required by aπ`Algo` to run protocol π are accessed through inherited operations of aπ`Object`; this is precisely where protocol composition occurs. So, when some operation related to protocol π is invoked on aπ`Object`, its execution is delegated to aπ`Algo`; symmetrically, when some operation on which protocol π relies is needed by aπ`Algo`, the latter invokes the corresponding (inherited) operation on aπ`Object`.

The way aπ`Object` and aπ`Algo` cooperate can be seen as some kind of *symbiosis*: two dissimilar objects participate to a mutually beneficial relationship. Instances of class π`Algo` each represent a distinct execution of protocol π's algorithm. So, another way to look at algorithm objects can be expressed as follows: at runtime, each algorithm object represents a layer in one of the protocol stacks currently in execution within aπ`Object`. This is presented in Figure 5 (b).

Because aπ`Object` must be able to execute all lower-level protocols on which protocol π is based, protocol programmers are forced to clearly specify what underlying protocols are required to implement corresponding protocol π; right side of Figure 5 (b) illustrates this point for object `anABCAlgo`. Only then, can protocol programmers choose which protocol class to subclass in order to implement π`Object`.

4.3 Flexible Composition

The BAST approach makes it possible to change any protocol algorithm of any protocol class very easily, e.g., one could modify the reliable multicast algorithm on which class `ABCObject` relies, while leaving it unchanged in other classes. Algorithms could even be chosen at runtime. Besides, dependencies between protocols remain only at the specification level (i.e., through subclass relationships depicted in Figure 2), whereas the implementations (i.e., algorithm objects) can vary. Another advantage is that protocol programmers can strictly focus on protocol π and on the interaction between class π`Object` and class π`Algo`: they need not to know how other protocols are implemented. In particular, all protocols needed to support protocol π are transparently used through inherited operations of class π`Object`. Such operations may also be based on one or more algorithm objects, but this is transparently managed by superclasses of π`Object`. A complete step-by-step methodology for extending BAST through protocol composition can be found in [6].



(a) Protocol and Algorithm Objects Interaction

(b) Algorithm Objects as Protocol layers

Figure 5: Modelling Protocol Algorithms as Objects

5 Performances

Our first prototype of the BAST library of protocol classes was implemented using VisualWorks 2.x, the commercial Smalltalk [7] platform by ParcPlace-Digitalk Inc. The development took place on a network of Sun SPARCstations 20 and Ultra 1, running Solaris 2.x operating system. It was recently ported to Java, using Sun's Java Development Kit (JDK) on the same hardware/OS environment.

Because Smalltalk and Java are "semantic twins", BAST's design is virtually the same in both languages. The only differences have to do with dynamic type checking in Smalltalk vs. static type checking in Java, and with their respective standard libraries. None of those differences are relevant here. In this section, we present some of our first performance results on Smalltalk.

5.1 Test Scenario

Our test scenario aims at testing BAST's performance in the context of one-to-many communication protocols. Three protocols are tested:

- a mere *sequence of reliable point-to-point sends*, through operation `rSend()` and callback `rDeliver()` of class `RMPObject`;
- a *reliable multicast*, through operation `rmcast()` and callback `rmDeliver()` of class `RMCObject`;
- an *atomic broadcast*, through operation `abcast()` and callback `abDeliver()` of class `ABCObject`.

In this scenario, we consider a logical ring composed of three protocol objects located on three different network nodes, with each protocol object o_i knowing its preceding object $pred(o_i)$. Each object o_i sends its messages to all protocol objects in the ring (including itself), using some one-to-many communications operations (depending on the protocol being tested). Messages are said to be "empty", i.e., they hold a reference to the `nil` object, and they are sent in bursts of five, i.e., when protocol object o_i sends messages to all, it does so five times in a row. To implement fair accesses to the transport layer, each o_i waits until it x`Deliver`s five messages from $pred(o_i)$ before sending another burst of five messages to all, with $x \in \{$`r, rm, abc`$\}$. At each network node, time is started when o_i x`Deliver`s its first message and stopped when it did so for 150 messages. By dividing 150 messages by the time o_i took to deliver them all, we get the throughput of each protocol for our scenario.

Testing took place on a 10 Mbit Ethernet interconnecting 13 Sun SPARCstations during a normal workday. All workstations were running XWindows as well as several interactive applications (netscape, emacs, etc.), so network and cpu loads were medium to high. Network nodes hosting protocols objects were three Sun UltraSPARC 1, all equipped with 64 MBytes RAM and running Solaris 2.5. Each test was performed at least 20 times to get meaningful results.

5.2 Performance Results

Figure 6 presents performance results for the scenario described above. The first column indicates which protocol is being tested, while the second column gives its throughput; performance worsen quickly when protocol complexity increases. Column 3 shows how many low-level messages are going through the transport layer in order to deliver 150 high-level messages; column 4 shows how the average size of messages evolves. Although high-level messages merely contain the `nil` object, lower-level messages contain additional information related to the various underlying protocols; this information gets bigger as protocols become more complex.

Not surprisingly, 300 messages are processed by `RMPObject` objects performing reliable point-to-point communications: 150 are sent and 150 are received. Reliable multicast objects of class `RMCObject` process roughly 4 times more messages to do their job: each message received for the first time triggers three more reliable point-to-point communications (because of the systematic retransmission scheme described earlier). For `ABCObject` objects performing atomic broadcasts, the number of messages is no more trivial to interpret: it includes many messages involved in underlying reliable multicast and consensus executions.

Column 5 gives the percentage of the overall time spent on marshaling/unmarshaling messages: those results strongly suggest that marshaling is where optimisation efforts should concentrate. Last column shows how many algorithm objects are created. Class `RMPObject` is the only one to directly rely on the transport layer. `RMCObject` objects only create `RMCAlgo` instances whereas `ABCObject`*s* also create consensus algorithms of class `CSSAlgo`. Out of 344 algorithm objects created, 53 are instances of class `CSSAlgo`, which indicates that so many consensus executions were necessary to `abDeliver()` 150 messages.

We also compared the performance of class `RMPObject`, which directly relies on our C transport layer `mpl`, implementing the scenario described above (same network configuration, message size, burst size, etc.). With such a C program, we get a throughput of 306 [msgs/sec] : The C transport layer alone is then about 22 times faster than class `RMPObject`. This can be partly explained by the bad performance of BOSS-based marshaling, but more tests will be necessary to refine our analysis and to further optimise BAST.

5.3 Possible Optimisations

As already suggested, optimisation should start at the marshaling level, in order to make it work faster and produce smaller messages. This result is not surprising: current BAST's marshaling is based on VisualWorks *Binary Object Storage Service* (BOSS), which aims at making it easy to save any object onto stable storage[4]. Although BOSS semantics suits our need for marshaling, it was designed with very different performance goals in mind.

However, before even touching BOSS internal code, a very simple optimisation principle can help us to cut down BAST's marshaling overhead: by reducing the number of low-level messages going through the transport layer. We applied this optimisation principle in two straightforward ways: (1) we avoid systematic retransmission at the reliable multicast level and we only retransmit messages if their initial sender is suspected to be faulty, (2) we avoid to unmarshal messages whenever an unused *ack* or *no ack* is received by the transport layer. With those two simple optimisations, we raised class `RMCObject` throughput from 4 to 12 [msgs/sec], while class `ABCObject` doubles its one. Further testing is now being carried on to determine where precisely should efficient optimisations occur in BOSS, in order to achieve faster marshaling. This issue is beyond the scope of this paper.

[4]In the Java implementation, we use a similar service.

6 Concluding Remarks

In this paper, we have presented BAST, an object-oriented library of reliable distributed protocols. BAST provides protocols like distributed total order broadcast and atomic commitment, supporting the development of reliable distributed applications. Further protocols, such as group membership or view synchronous communication, can easily be developed, by composing lower-level protocols provided by BAST, such as failure detection, reliable multicast and consensus.

We described the way BAST was designed, and we pointed out some issues in reusing and composing distributed protocols. In particular, we described the way BAST enables concurrent executions of different instances of the same protocol.

As mentioned earlier, BAST was primarily built as an *open* environment for teaching and prototyping reliable distributed algorithms. In particular, all optimisations that would compromise its extensibility and genericity were left out, e.g., allowing *any* object to be sent across the network has a cost, and as shown in our performance figures, marshalling and unmarshalling is the main source of inefficiency. We are currently achieving in-depth performance analysis in order to seek ways of optimising it.

More information about BAST can be found on the web, at *http://lsewww.epfl.ch/bast*. Our first BAST implementation is public-free and is available there.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Welsey, 1987.

[2] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.

protocol class	**throughput** [msgs/sec]	**number of mpl-msgs**	**mpl-msg size** [bytes]	**marshaling** [time percentage]	**algorithms created**
`RMPObject`	14	300	700	61	*none*
`RMCObject`	4	1173	1334	92	150
`ABCObject`	1	3406	2045	80	344

Figure 6: Performance when x`Deliver`*ing* 150 messages, $x \in \{$`r, rm, abc`$\}$

[3] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267, March 1996.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] B. Garbinato, P. Felber, and R. Guerraoui. Modeling protocols as objects for structuring reliable distributed systems. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'97)*, pages 165–171, Phoeniz, Arizona, January 1997.

[6] B. Garbinato and R. Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In *Proceeding of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 221–232, Portland (Oregon), June 1997.

[7] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison-Wesley, 1983.

[8] R. Guerraoui. Modular atomic objects. *Theory and Practice of Object Systems*, 1(2):89–100, 1995.

[9] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: Bridging the gap. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 121–132. Springer Verlag, 1995.

[10] H. Hüni, R. Johnson, and R. Engel. A framework for network protocol software. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*. ACM Press, 1995. Special Issue of Sigplan Notices.

[11] S. Mishra, L.L. Peterson, and R.D. Schlichting. Experience with modularity in Consul. *Software Practice and Experience*, 23(10):1053–1075, October 1993.

[12] G. Parrington and S. Schrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'88)*, Norway, August 1988.

[13] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abott. RPC in the x-Kernel: Evaluating new design techniques. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 91–101, November 1989.

[14] S. Popovitch, G. Kaiser, and S. Wu. An object-based approach to implementing distributed concurrency control. In *IEEE Conference on Distributed Computing Systems Proceedings*, pages 65–72, Arlington (Texas), May 1991.

[15] M.T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[17] D.C. Schmidt. ADAPTIVE: A dynamically assembled protocol transformation, integration and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[18] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'95)*, August 1995.