Strategy Pattern for Composing Reliable Distributed Protocols

Benoît Garbinato Pascal Felber Rachid Guerraoui

Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Suisse
e-mail: bast@lse.epfl.ch

Abstract

Reliable distributed systems involve many complex protocols. In this context, *protocol compositions* is a central concept, because it allows to reuse robust protocol implementations. In this paper, we describe how the *Strategy* pattern has been recursively used to support protocol composition in the BAST framework. We also discuss some design alternatives, some of which have been applied in other existing frameworks.

1 Introduction

This paper presents how the Strategy pattern has been used to build BAST[1], an extensible object-oriented framework for programming reliable distributed systems. Protocol composition plays a central role in BAST and relies on the notion of protocol class. In this paper, we focus on the recursive use of the Strategy pattern to overcome the limitations of inheritance, when trying to flexibly compose protocols. In a companion paper [6], we have presented how generic agreement protocol classes can be customized to solve the atomic commitment and the total order multicast, which are central problems to transactional systems and to group-oriented systems respectively.

The BAST framework

Building reliable distributed systems is a challenging task, because one has to deal with many complex issues, e.g., reliable communications, failure detections[2], distributed consensus, replication management, transactions management, etc. Each of these issues corresponds to some distributed protocol and there are many. In such a protocol "jungle", programmers have to choose the right protocol for the right need. Besides, when more than one protocol is necessary, comes the problem of their interactions, which further complexifies programmers' task.

The BAST framework aims at structuring reliable distributed systems by allowing complex distributed protocols to be composed in a flexible manner. For example, by adequately composing reliable multicast protocols with transactional protocols, BAST makes it possible to transparently support transactions on groups of replicated objects. It relies heavily on the *Strategy* pattern, which is recursively used to get around the limitations of inheritance as far as protocol composition goes. Our current implementation has been written in Smalltalk [7] and is fully operational. It is intended to be used as a prototyping platform for new reliable distributed protocols. This will help us to further test our approach, by adding more and more protocol classes. Currently, a C++ version of the BAST framework is being worked on in the context of the *Opendreams* project (ESPRIT n°29843), which aims at designing a *reliable* CORBA compliant platform. We also plan to port BAST to the Java [8] program-

[1] The BAST framework is part of *Phoenix*, an on-going research project at the Operating Systems Laboratory of the Swiss Federal Institute of Technology, Lausanne.

[2] The concept of failure detection enables to hide the timeouts commonly used in distributed systems [2].

ming environment.

Overview of the paper

Section 2 introduces the concept of protocol object as defined in BAST, and how it helps to structure distributed systems and to deal with failures. Section 3 discusses why inheritance alone is limited in supporting flexible protocol composition and presents how we applied the Strategy pattern to break these limitations. We also show how the Strategy pattern is transparently used in a recursive manner, and we present what steps have to be performed in order to extend BAST through protocol composition. Section 4 discusses various design alternatives, and compares our approach with other research works described in the literature. Finally, section 5 summarizes what has been presented in this paper, as well as the future we are planning for the BAST framework.

2 Protocol objects

The BAST framework[3] was designed to help programmers in building reliable distributed systems, and is based on protocols as basic structuring components. With BAST, a distributed system is composed of *protocol objects* that have the means to remotely designate each other and to participate in various protocols, whereas *distributed protocol* π is a set of interactions between protocol objects that are aiming at solving *distributed problem* π. We use `a`π`Object` to name a protocol object capable of participating in protocol π, and we say that π`Object` is its *protocol class*. Each π`Object` provides a set of operations that interface protocol π, i.e., these operations act as entry points to the protocol. Abstract class `Protobject` is the root of the protocol class hierarchy.

With such broad definitions, any interaction between objects located on distinct network nodes is a protocol, even a mere point-to-point communication. For example, class `RMPObject` implements a reliable point-to-point communication protocol and provides operations `rSend()` and `rDeliver()` that enable to reliably send, respectively receive, any object[4]; operation `rDeliver()` is a redefinable callback and is said to be triggered by the protocol. Note that such an homogeneous view of what distributed protocols are does not contradict the fact that some protocols are more basic than others. Communication protocols, for example, are fundamental to almost any other distributed protocol.

Dealing with failures. Because failures are part of the real world, there is the need for *reliable* distributed protocols e.g., consensus, atomic commitment, total order multicast. Reliable distributed protocols are challenging to implement because they imply complex relationships with other underlying protocols. For example, both the atomic commitment and the total order multicast rely on consensus, while the latter is itself based on failure detections, on reliable point-to-point communications, and on reliable multicasts. In turn, reliable multicasts can be built on top of reliable point-to-point communications. Figure 1 (a) presents an overview of some distributed protocol dependencies.

In BAST, protocol classes are organized into a single inheritance hierarchy which follows protocol dependencies, as pictured in figure 1 (b). Each protocol class implements only one protocol, but instances of some π`Object` class can execute any protocol inherited from π`Object`'s superclasses. Protocol objects are able to run several executions of identical and/or distinct protocols concurrently.

3 Strategy pattern in BAST

Composing protocols

With protocol objects, managing protocol dependencies is not only necessary during the design and implementation phases (between protocol classes), but 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. In this context, trying to compose protocols comes down to answering the question *"How are protocol layers assembled and how do they cooperate?"*.

Figure 2 (a) presents a runtime snapshot of `aCSSObject`, some protocol object of class `CSSObject` which implements an algorithm for solving the distributed consensus problem. The consensus problem is defined on some set σ of distributed objects as follows: all correct objects in σ propose an initial value and must reach agreement

[3] We named BAST after the cat-goddess of the Egyptian mythology.

[4] We mean here any object that is *not* a protocol object. Allowing to send protocol objects across the network implies to solve the distributed object migration problem.

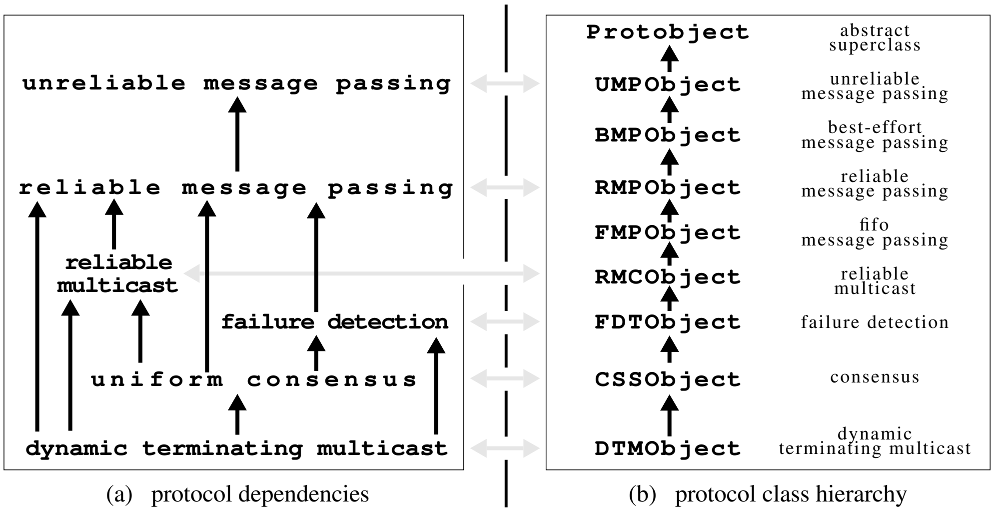


Figure 1: Protocols and protocol classes in Bast

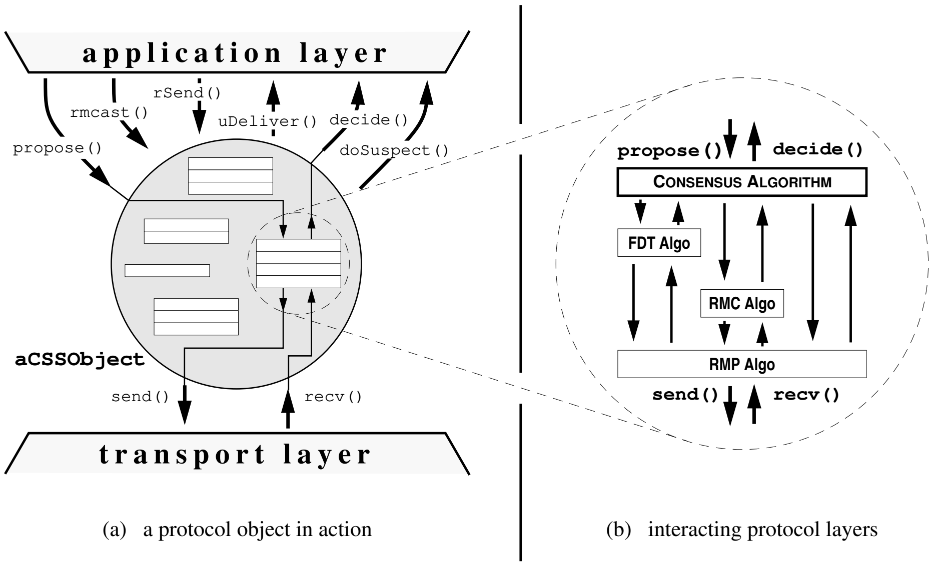
on one of the proposed values (the decision) [3]. Class `CSSObject` defines operations `propose()` and `decide()`, which mark the beginning and the termination of the protocol respectively [2]. Besides consensus, protocol object `aCSSObject` is also capable of executing any protocol inherited by its class, e.g., reliable point-to-point communications and reliable multicasts, as well as failure detections. In figure 2 (a), `aCSSObject` is concurrently managing five different protocol stacks for the application layer, and issuing low-level calls to the transport layer. Focusing on the consensus stack, protocol composition means here to assemble various layers, each being necessary to execute the consensus protocol, into the protocol stack pictured in figure 2 (b). The assembling occurs at runtime and creates a new stack each time the application invokes operation `propose()`.

Inadequacy of inheritance alone. With Bast, distributed applications are structured according to their needs in protocols: they are made of protocol objects, which act as distributed entities capable of executing various protocols. With this approach, it all comes down to choosing the right class for the right problem. We believe that inheritance is an appropriate tool to achieve this: by passing appropriate arguments to protocol operations and by implementing callback operations, programmers have the ability to tailor generic protocol classes to their needs. However, we pretend that 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. Furthermore, inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*. These limitations lead us to seek an alternative solution for flexible protocol composition.

Protocol algorithms as strategies

According to Gamma and al., the intent of the *Strategy* pattern is to "*define a family of algorithms, encapsulate each one, and make them interchangeable*" [5, page 315]. This is usually achieved by objectifying the algorithm [4], i.e., by encapsulating it into a so-called *strategy* object; the latter is then used by a so-called *context* object. Making each π`Object` protocol class independent of the algorithm supporting protocol π is precisely what we need to be able to compose reliable distributed protocols in a flexible manner.

In the Bast framework, strategy objects represent protocol algorithms and they are instances of subclasses of class `ProtoAlgo`. A `ProtoAlgo` subclass that implements an algorithm for solving problem π is referred to as class π`Algo`. In the Strategy pattern terminology, a protocol algorithm, instance of some π`Algo` class, is a *strategy*, and a protocol object, instance of some π`Object` class, is a *context*. A strategy and its context are strongly cou-



(a) a protocol object in action (b) interacting protocol layers

Figure 2: Protocol layers and protocol objects

pled and the application layer only deals with instances of π`Object` classes, i.e., it knows nothing about strategies.

Strategy/Context interactions. Figure 3 (a) sketches the way protocol objects and algorithm objects interact. On the left side, protocol object aπ`Object` offers the services it inherits from its superclasses, as well as the new services that are specific to protocol π. The actual algorithm implementing protocol π is not part of aπ`Object`'s code: instead, the latter uses services provided by strategy aπ`Algo` (on the right side of figure 3 (a)). Whenever an operation related to protocol π is invoked on aπ`Object`, the execution of the protocol is delegated to strategy aπ`Algo`. In turn, the services required by the strategy to run protocol π are based on the inherited services of context aπ`Object`. Such required services merely identify entry point operations to underlying protocols needed to solve problem π.

Each instance of class π`Algo` represents one execution of protocol π implemented by that class, and holds a reference to the context object for which it is running; any call to the services required by the strategy will be issued to its context object. There might be more than one instance of the same `ProtoAlgo`'s subclass used simultaneously by aπ`Object`. At runtime, the latter maintains a table of all strategies that are currently in execution for it. Each message is tagged to enable aπ`Object` to identify in which execution of what protocol that message is involved, and to dispatch it to the right strategy. Figure 3 (b) presents the relationship between classes π`Object` and π`Algo`, using a class diagram based on the OMT (Object Modeling Technique) notation [15]. The correspondence between π`Algo` strategy objects and layered protocol stacks is pictured in figure 3 (c): at runtime, each strategy object represents a layer in one of the protocol stacks currently in execution.

The context/strategy separation enables to overcome the limitations of inheritance, as far as protocol composition goes. One could for example optimize the reliable multicast algorithm and use it in some protocol classes, while leaving it unchanged in others. Different protocol algorithms could even be chosen dynamically, according to some criteria computed at runtime. This approach also helps protocol programmers to clearly specify, for each protocol π, what are its dependencies with other protocols. One drawback of the Strategy pattern is the overhead due to local interactions between strategies and contexts. In distributed systems however, this overhead is small compared to communication delays, especially when failures and complex protocols are involved. More specifically, the time for a local Smalltalk invocation is normally under 100 μs, whereas a reliable multicast communication usually takes more than 100 ms when three or more protocol objects are involved[5] (without even considering

[5]Those values were measured on a network of Sun

failures). The gain in flexibility clearly overtakes the local overhead caused by the use of the Strategy pattern.

Reliable multicast: an example

We now present how we implemented reliable *multicast* communications using the Strategy pattern. In BAST, class `RMCObject` provides primitives `rmcast()` and `rmDeliver()` that enable to send, respectively receive, a message `m` to a set of protocol object references `destSet`, in a way that enforces reliable multicast properties. Current implementation of class `RMCObject` relies on strategy class `RMCAlgo`.

Overview of the protocol. The protocol starts when operation `rmcast()` is invoked on some initiator object $\texttt{aRMCObject}_i$, passing it a message `m` and a destination set `destSet`. In this operation, context $\texttt{aRMCObject}_i$ first creates a strategy $\texttt{aRMCAlgo}_i$, and then invokes operations `rmcast()` on it, with the arguments it just received. Strategy $\texttt{aRMCAlgo}_i$ builds message $\tilde{\texttt{m}}$, containing both `m` and `destSet`. It then issues a reliable point-to-point communication with each protocol object referenced in `destSet`; in order to do this, strategy $\texttt{aRMCAlgo}_i$ relies on inherited service `rSend()` of context $\texttt{aRMCObject}_i$. When message $\tilde{\texttt{m}}$ reaches $\texttt{aRMCObject}_t$, one of the target objects, operation `rDeliver()` is triggered by the protocol. Operation `rDeliver()` detects that $\tilde{\texttt{m}}$ is a multicast message and forwards it to $\texttt{aRMCAlgo}_t$, the strategy in charge of that particular execution of the reliable multicast protocol. When $\texttt{aRMCAlgo}_t$ receives $\tilde{\texttt{m}}$ for the first time, it re-issues a reliable point-to-point communication with each protocol object referenced in `destSet` (extracted from $\tilde{\texttt{m}}$), and then invokes `rmDeliver()` on its context $\texttt{aRMCObject}_t$, passing it message `m` (also extracted from $\tilde{\texttt{m}}$). This retransmission scheme is necessary because of the *agreement* property of the reliable multicast primitive, which requires that either all correct objects in `destSet` or none receive message `m` [2].

Recursive use of Strategy pattern

When solving distributed problem π, one can strictly focus on the interaction between class π`Object` and class π`Algo`, while forgetting about how other protocols are implemented. In particular, all protocols needed to support protocol π are transparently used through inherited services of class π`Object`. Those services might also be implemented applying the Strategy pattern, but this is transparently managed by inherited operations of π`Object`. In that sense, BAST uses the Strategy pattern in a powerful *recursive* manner.

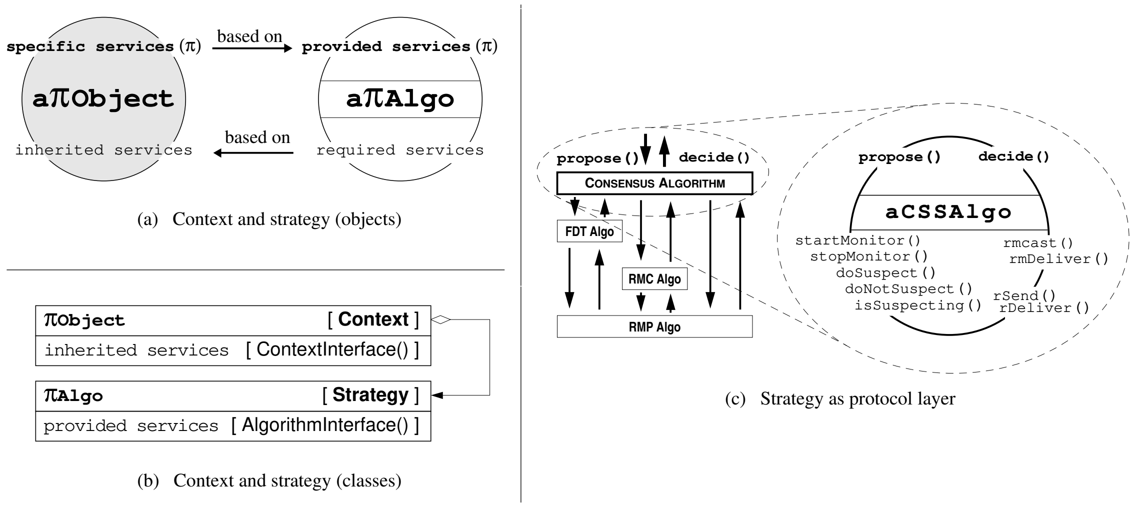
The recursive use of the Strategy pattern is illustrated in figure 4. The latter schematically presents a possible implementation of protocol class `CSSObject` presented in section 3, which enables to solve the distributed consensus problem by providing operations `propose()` and `decide()`. On this figure, the gray oval is context class `CSSObject`, while inner white circles are various π`Algo` strategy classes (π being different protocols). Arrows show the connections between provided services (top) and required services (bottom) of each strategy class. Operations provided by class `CSSObject` are grouped on the application layer side (top). Each strategy class pictured in figure 4 is managed by the corresponding context class in the protocol class hierarchy presented in figure 1 (b).

Extending the BAST framework

Basing the BAST framework on the Strategy pattern has the advantage to make it easily extensible. To illustrate this, we now present how we built `DTMObject`, a protocol class supporting the *Dynamic Terminating Multicast* (DTM) protocol from existing contexts and strategies. The DTM protocol can be understood as a common denominator of many reliable distributed algorithms [9].

Overview of the protocol. The protocol starts by the invocation of operation `dtmcast()` on an initiator object, passing it a message `m` and a set of protocol object references `destSet`. This invocation results in a reliable multicast of `m` to the set of participants objects. When message `m` reaches some participant, the protocol triggers operation `receive()`, passing `m` as argument. The participant object then computes a reply and returns it. Eventually, operation `interpret()` is triggered by the protocol on each non-faulty participant object, taking `replySet`, a subset of the participants' replies, as argument. The protocol insures that all correct participant objects get the same subset of replies, i.e., a consensus has been reached on that set.

SPARCstation 20 interconnected through a 10 Mbits Ethernet.



(a) Context and strategy (objects)

(b) Context and strategy (classes)

(c) Strategy as protocol layer

Figure 3: Strategy pattern in Bast

Methodology for extending Bast**.** A five steps methodology guides programmers in extending the Bast framework using the Strategy pattern. We illustrate each of these steps below, by presenting how the methodology was applied to the design of class `DTMObject`.

1. Establish what services the new protocol class `DTMObject` provides, i.e., what operations are given to programmers wanting to use `DTMObject`; those operations are `dtmcast()`, `receive()` and `interpret()`.

2. Choose an algorithm implementing DTM and determine what services it requires, by decomposing it in a way that allows to reuse of as many existing protocols as possible; those services are: consensus, failure detections, reliable point-to-point and reliable multicast communications (see [9] for algorithmic details).

3. Implement the chosen algorithm in some `DTMAlgo` class; all calls to the above required services are issued to an instance variable representing the context object, i.e., an instance of class `DTMObject`.

4. Choose the protocol class that will be derived to obtain new class `DTMObject`; the choice of class `CSSObject` is directly inferred from step 2, since the chosen superclass has to provide *at least* all the services required by protocol DTM.

5. Implement class `DTMObject` by connecting services provided by class `DTMAlgo` to new DTM-specific services of class `DTMObject`, and by connecting services required by class `DTMAlgo` to corresponding inherited services of class `DTMObject`.

4 Design alternatives

Our first implementation of Bast was not based on the Strategy pattern, i.e., distributed algorithms were not objects, and protocol objects were not capable of participating in more than one protocol execution concurrently. Furthermore, protocol composition was only possible through *single* inheritance[6].

Because protocol objects are the basic addressable distributed entities in our approach, it is not possible to guarantee that there will never be more than one protocol execution involving each protocol object at a given time. For example, we cannot make sure that there will not be two concurrent multicast communications and/or transactions involving the same protocol objects. Allowing concurrency at this level is an essential feature. And as far as protocol composition is concerned, single inheritance is way beyond offering a satisfactory degree of flexibility.

For all these reasons, we made Bast evolve through a second implementation which main goal was to overcome the limitations mentioned above. We now discuss some design alternatives that were considered in the process of implementing this second pro-

[6] Remember that we used Smalltalk as implementation language for prototyping.

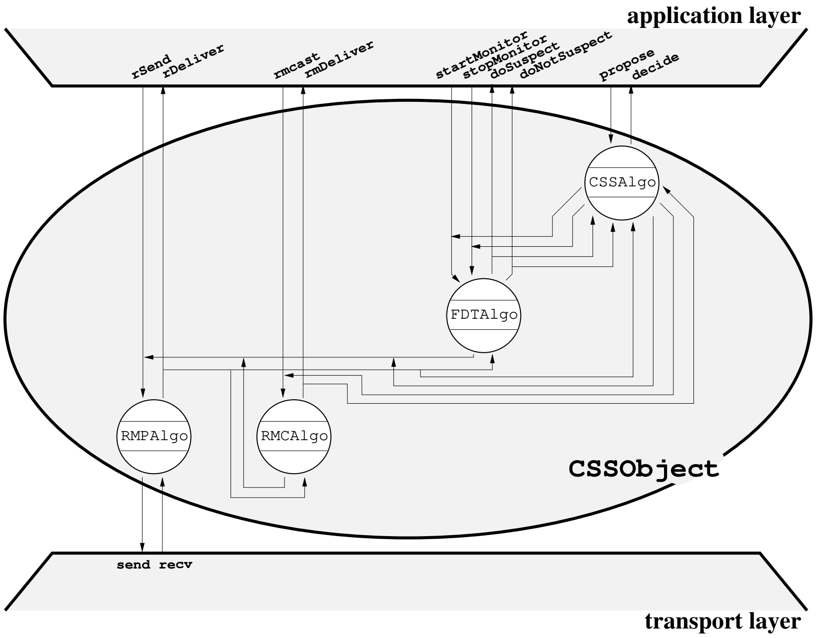


Figure 4: Recursive use of the Strategy pattern

totype of BAST, together with design issues that we studied from other existing frameworks described in the literature.

Multiple inheritance

Although our prototyping language does not offer multiple inheritance, assembling the various protocol layers through this code reuse mechanism is very appealing[7]. The idea is to make each protocol class π`Object` implement only protocol π, while accessing all required underlying protocols through unimplemented operations; each protocol class is then an *abstract* class. Before being able to actually instantiate a protocol object, one first has to build a new class deriving from all the necessary protocol classes.

There are two major drawbacks with this approach. First, protocol layers can only be assembled through subclassing, and it is thus difficult to compose protocol at runtime if not impossible (e.g., in C++ classes are only compile-time entities). Second, we still have to manage concurrent protocol executions within the same protocol object, while this problem is handled nicely as soon as algorithms are manipulated as objects.

[7] Ingalls and Borning have shown how reflective facilities of Smalltalk can be applied to extend the language with multiple inheritance [11], so we could have used that technique if we really wanted to.

Toolbox approach

Another possible approach to reuse protocol implementations is to provide programmers with a toolbox containing reusable components and associate them with design patterns. Both ASX [16] and CONDUITS+ [10] frameworks can be seen as such toolboxes. The ASX framework provides collaborating C++ components, also known as wrappers, that help producing reusable communication infrastructures. These components are designed to perform common communication-related tasks, e.g., event demultiplexing, event handler dispatching, connection establishment, routing, etc. Several design patterns, such as the *Reactor* pattern and the *Acceptor* pattern, act as architectural blueprints that guide programmers in producing reusable and portable code. In CONDUITS+ [10], two kinds of objects are basically offered: *conduits* and *information chunks*, which can be assembled in order to create protocol layers and protocol stacks. Various patterns are also provided to help programmers in building protocols.

However, there is no such thing as protocol object in either of the above frameworks. Since our main intend is to provide programmers with a powerful unifying concept, the *protocol object*, we did not choose a toolbox approach for BAST. Furthermore, ASX does not promote protocol composition, whereas CONDUITS+ does it in a slightly different way than BAST, as we shall see now.

Black-box framework

CONDUITS+ offers basic elements that helps programmers to build protocol layers. The use of design patterns is motivated by the fact that traditional layered architecture do not allow code reuse across layers, which is precisely what CONDUITS+ aims at. Protocols can then be composed with CONDUIT+, but at lower-level than BAST, i.e., through the assembling of conduits and information chunks, which are elementary blocks used to build protocol layers. In other words, the CONDUIT+ framework does not allow to manipulate protocol layers as objects, but only *pieces* of protocol layers. Compared to BAST, protocol algorithms are further decomposed in CONDUIT+: conduits and information chunks are finer grain objects than BAST's strategies. Indeed, the latter represent protocol layers, while the former are internal components of protocol layers. CONDUIT+ goes one step further in the process of objectifying protocol algorithms.

This approach makes it easy for CONDUIT+ to be a pure *black-box* framework, while BAST combines features of both *black-box* and *white-box* frameworks[8]. With BAST, we are considering to definitively get rid of inheritance but this issue has to be carefully studied, because it would have important consequences on the way BAST can be used.

Modeling communications

Several systems try to model communications but do not address reliability issues, e.g., STREAMS [14] and the x-Kernel [13]. Other systems offer reliable distributed communications, either based on groups as elemental addressing facilities, e.g., CONSUL [12], ISIS [1] and HORUS [18], or based on transactions, e.g., ARJUNA [17]. Most of these systems merely provide a non-extensible library of communication protocols.

With BAST, we have tried to model *any kind* of interaction between distributed object, not only communications. This is necessary if we want to deal with failures in an extensible way, because reliable protocols tend to become much more complex than mere communications. As far as protocol composition is concerned, the HORUS system enables to build protocol stacks from existing layers in a strictly vertical manner. However, it is based on groups as fundamental addressing facility and provides no framework and/or pattern for building *new* protocols layers. By making protocol objects BAST's basic distributed entities, we can build both the group model and the transaction model [6]. Furthermore, the Strategy pattern provides a powerful scheme for creating new protocols through composition.

5 Concluding remarks

In this paper, we presented how protocol objects can help to build reliable distributed systems. We focused on how the Strategy pattern enables to overcome the limitations of inheritance, when trying to compose protocols. As far as we know, BAST is the only framework providing the concept of protocol object, so we see it as our contribution to the design of well-structured reliable distributed systems. Our current prototype of BAST is fully operational. At the moment, inheritance is still partly involved when composing distributed protocols; although a minor drawback, this does not make protocol composition as flexible as one could expect. This is due to the fact that programmers have to know something about the implementation of the protocol classes they reuse, namely their inheritance relationships. It is not surprising, since inheritance is known to violate encapsulation and to hinder modularity. Future work will consist in trying to decide if getting definitively rid of inheritance, at least as far as protocol composition goes, is a good way to achieve even more flexibility. We also plan to extend BAST with new protocol classes, supporting frequently used protocols in reliable distributed systems.

[8] In a *black-box* framework, reusability is mainly achieved by assembling instances, whereas in a *white-box* framework, it is mainly achieved through inheritance. A black-box framework is easier to use, but harder to design.

References

[1] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[2] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR94-1458, Cornell University, Computer Science Department, October 1994. A preliminary version appears in PODC'91.

[3] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical report, Department of Computer Science, Yale University, June 1983.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1993.

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

[6] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*. Springer Verlag, July 1996.

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

[8] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, Inc., October 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] D.H.H. Ingalls and A.H. Borning. Multiple inheritance in smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237. AAAI, 1982.

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

[13] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abott. Rpc in the $x-$Kernel: Evaluating new design techniques. *ACM Symposium on Operating Systems Principles*, 23(10):91–101, November 1989.

[14] D. Ritchie. A stream input-output system. *Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[16] D.C. Schmidt. ASX: an object-oriented framework for developing distributed applications. In *Proceedings of the 6^{th} USENIX C++ Technical Conference*. USENIX Association, April 1994.

[17] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 1991.

[18] R. van Renesse and K. Birman. Protocol composition in Horus. *ACM Principles of Distributed Computing*, 1995.