Using the Strategy Design Pattern to Compose 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

When building *reliable* distributed systems, one has to deal with many complex protocols, e.g., failure detection, distributed consensus, atomic commitment, and being able to reuse robust protocol implementations based on well-tested code is a key feature. In this context, *flexible composition of reliable distributed protocols* is a central concept, because it allows to customize complex protocols and to build new protocols from existing ones. This paper describes how the *Strategy* design pattern has been extensively used to support protocol composition, where inheritance alone was not adequate.

1 Introduction

This paper presents how the Strategy design 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 use of the Strategy design pattern to overcome the limitations of inheritance, when trying to flexibly compose protocols. In a companion paper [8], 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.

[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.

Composing distributed protocols

Object paradigms have deeply affected the way people design distributed systems. The CORBA standard [12] is a good example of the interest in object paradigms from the distributed systems community. In CORBA, specialized abstractions are defined in order to support distributed objects. Other systems support distributed programming through either fully reflexive architectures [1, 17] or limited reflexive facilities [9, 2]. As far as reliability issues go, transactional systems, such as ARJUNA [15], and group-oriented systems, such as GARF [9], are often two alternative ways of achieving reliability in distributed systems. Blending them is very difficult, because of their different underlying model.

Why compose protocols? Building reliable distributed systems is a challenging task, because one has to deal with many complex issues, e.g., reliable communications, failure detections, 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. It does so 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.

Why use design patterns? Although a fairly recent emerging trend, design patterns have already been applied to many research areas, in particular to object-oriented distributed systems. This can be easily explained by the inherent complexity of building such systems: since design patterns enable to capture and transmit recurring object-oriented solutions [4], they are powerful tools to encapsulate the knowhow in distributed system design. The BAST framework relies heavily on the *Strategy* pattern, which is recursively used to get around the limitations of inheritance as far as protocol composition goes.

Status of the BAST framework. BAST's first implementation is fully operational. This prototype has been written in Smalltalk [10], and 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 [11] programming environment.

Overview of the paper

Section 2 presents an overview of BAST's protocol class hierarchy and discusses why inheritance alone is inadequate to support flexible protocol composition. Section 3 introduces the Strategy pattern and explains how we applied it to break the limitations due to inheritance. 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 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 Context and problem

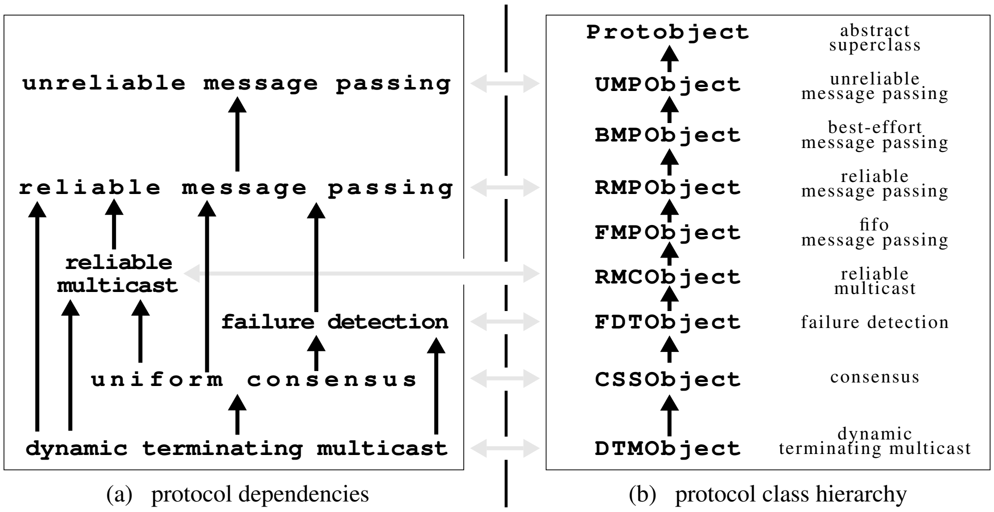
A distributed system can be viewed as a set of *distributed objects* that communicate by message passing, i.e., they have the means to remotely designate each other, and to send messages to each other. Distinct distributed objects reside *a priori* on different nodes, and faulty objects are supposed to fail independently by crashing. No assumption is made on the synchrony of the system, but the notion of failure detector is used instead[2]. We define *distributed protocol* π as a set of coordinated interactions between distributed objects, which are aiming at solving *problem* π. This definition implies a very broad view of what distributed protocols are, since it all depends on the specification of problem π: even a trivial unreliable communication between two remote objects is seen as a distributed protocol. Because failures are part of the real world, there is the need for reliable distributed systems capable of hiding failures, i.e., of enforcing the *liveness* of the system, and capable of preventing failures from putting it in a inconsistent state, i.e., of enforcing the *safety* of the system. Reliable distributed protocols are the cornerstone of such systems, e.g., atomic commitment protocols play a central role in transactional systems, while reliable total order protocols are essential to both transactional and group-oriented systems. Reliable distributed protocols are often challenging to implement, because they involve many other underlying protocols. This results in complex dependency relationships. The consensus protocol, for instance, is based on a failure detection protocol, on a reliable point-to-point communication protocol, and on a reliable multicast protocol. In turn, the latter also relies on a reliable point-to-point communication protocol. Figure 1 (a) presents an overview of some distributed protocol dependencies.

Protocol classes in BAST

The BAST[3] framework was designed to help programmers in building reliable distributed systems. Its approach is based on protocols as basic structuring components: in BAST, distributed protocols are manipulated as *classes of objects*. The concept of *protocol class* can be defined as follows: a protocol class π`Object` implements the behavior of distributed objects capable of executing protocol π. Instances of protocol classes are sometimes designated as *protocol objects*. In this paper, class π`Object` is used to refer to any protocol class, and aπ`Object` is used to designate some distributed in-

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

[3] We named the BAST framework after the cat-goddess of the Egyptian mythology. Since cats are said to benefit from several lives, which allow them to resist to a fair number of failures, we decided that the name of their protectress would be perfect for our reliable distributed framework.



(a) protocol dependencies (b) protocol class hierarchy

Figure 1: Protocols and protocol classes in BAST

stance of class π`Object`. In BAST, the root of the protocol class hierarchy is abstract superclass `Protobject`. Each π`Object` class provides a set of operations that interface protocol π, i.e., these operations act as entry points to the protocol for applications that use it. For example, class `RMPObject`, which implements a reliable point-to-point communication protocol, provides operations `rSend()` and `rDeliver()` that enable to reliably send, respectively receive, a message. In BAST, distributed protocols are said to be *generic*, which means that the type of arguments passed to any protocol operation is not restricted; e.g., operation `rSend()` can be used to send *any* object. Genericity also means that 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 organized into a single inheritance hierarchy which follows protocol dependencies (see 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.

The approach based on protocol classes makes it easier to reuse complex distributed algorithms and enables to build protocols in an incremental way. New protocols are created with minimal effort, while customizations and optimizations of existing ones are easily achieved through subclassing. This approach also promotes a modular decomposition of distributed protocols, which helps to better understand their differences, similarities and dependencies. Making different protocols work together is then made easier.

Composing protocols

As mentioned, relationships between protocols can be quite complex. Since we want to build reliable distributed systems, the intrinsic robustness of protocol implementations is an important issue for achieving reliability; so, reusing well-tested code is a key feature. In BAST, 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 the snapshot of an instance of class `CSSObject` during its execution; this class implements a protocol algorithm solving the 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 on one of the proposed values (the decision) [5]. More specifically, the consensus problem is specified in terms of primitives *propose()* and *decide()*, which mark the beginning and the termination of the protocol respectively [3];

class `CSSObject` defines corresponding operations `propose()` and `decide()`. Besides consensus related operations, protocol object `aCSSObject` is also capable of executing any protocol inherited by its class, e.g., reliable point-to-point communication and reliable multicast protocols, as well as failure detection protocols. In figure 2 (a), `aCSSObject` is concurrently running five different protocols for the application layer, and issuing low-level calls to the transport layer. Conceptually, each execution triggered by the application layer is supported by an independent protocol stack, as suggested in figure 2 (a). Furthermore, each layer of that stack is in charge of only one protocol supported by class `CSSObject`. Focusing on consensus, 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 for composing protocols. BAST's protocol class hierarchy enables to structure distributed systems according to their needs in protocols: such systems are made of protocol objects, which act as distributed entities capable of executing various protocols, depending on their class. With this approach, it all comes down to choosing the right class for the right problem, and to customize it. We believe that inheritance is an appropriate tool to achieve this: by passing appropriate arguments to protocol operations and by implementing callback operations according to their semantics, 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.

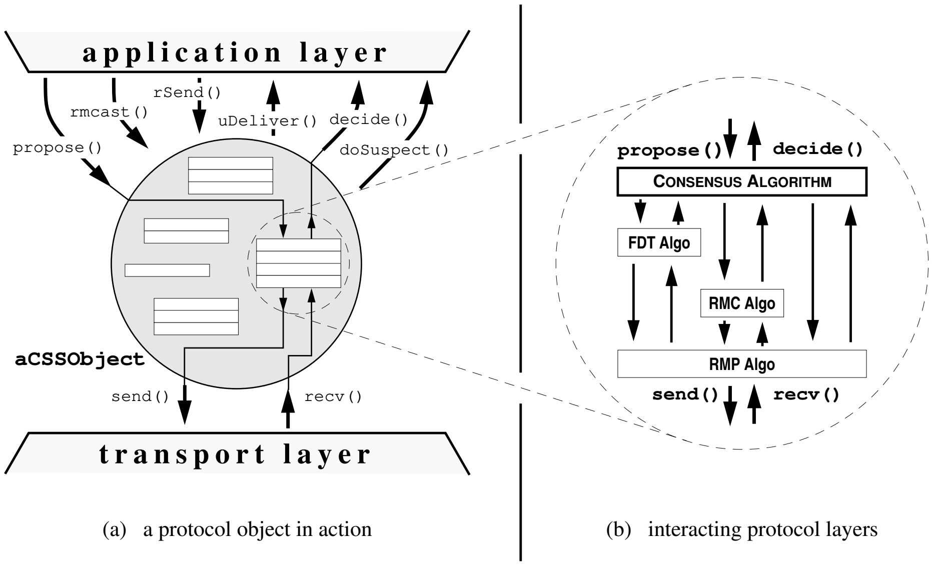
3 Strategy pattern in BAST

According to Gamma and al., the intent of the *Strategy* design pattern is to "*define a family of algorithms, encapsulate each one, and make them interchangeable*" [7, page 315]. This is usually achieved by objectifying the algorithm [6], 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.

Protocol algorithms as strategies

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 protocol 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 coupled 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 protocol algorithms interact. On the left side, protocol object `a`π`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 `a`π`Object`'s code: instead, the latter uses services provided by strategy `a`π`Algo` (on the right side of figure 3 (a)). Whenever an operation related to protocol π is invoked on `a`π`Object`, the execution of the protocol is delegated to strategy `a`π`Algo`. In turn, the services required by the strategy to run protocol π are based on the inherited services of context `a`π`Object`. 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 that context object. There might be more than one instance of the same `ProtoAlgo`'s subclass used simultaneously by the same protocol object. At runtime, `a`π`Object` maintains a table of all strategies that are currently in execution for him. Each message is tagged to enable `a`π`Object` to identify in which execution of what protocol that message is involved, and to dispatch it to the right strategy. Figure 3 (b) illustrates the relationship between π`Algo` strategy objects and layered protocol stacks: at runtime, each strategy object represents a layer in one of the pro-



(a) a protocol object in action

(b) interacting protocol layers

Figure 2: Protocol layers and protocol objects

tocol stacks currently in execution.

The context/strategy decomposition forces protocol programmers to clearly specify for each protocol π what are the dependencies with other protocols. Those dependencies then become the required services of class π`Algo`. This approach also has the advantage to overcome the limitations of inheritance, as far as protocol composition goes. One could for example optimize the reliable multicast strategy of some `CSSObject`'s subclass, while leaving it unchanged in class `CSSObject` itself. Different strategies could even be chosen during program execution, according to some criteria computed at runtime. One drawback of the Strategy pattern is the overhead due to local interactions between strategies and contexts. However, in distributed systems subject to failures, this overhead is small compared to communication delays, especially when failures and complex protocols are involved. We believe that the gain in flexibility overtakes the local overhead, at least during the design and prototyping phases.

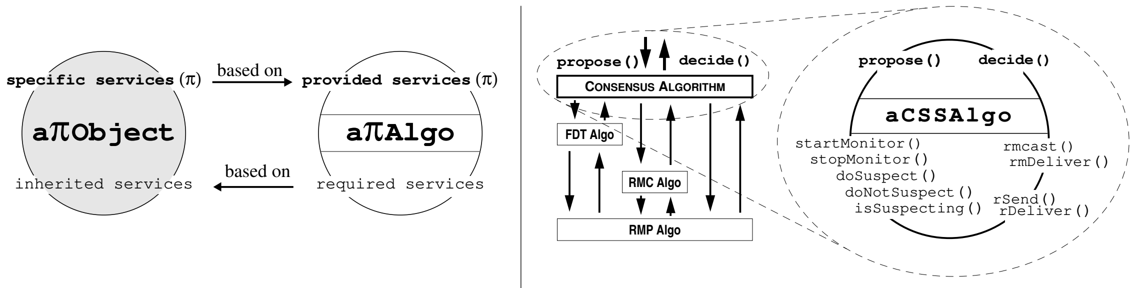
Reliable multicast strategy: example

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 objects `destSet` in a way that enforces reliable multicast properties. Current implementation of class `RMCObject` relies on strategy class `RMCAlgo` that implements a naive protocol.

Overview of the protocol. The protocol starts when operation `rmcast()` is invoked on some initiator object `aRMCObject`$_i$, passing it a message `m` and a destination set `destSet`. In this operation, context `aRMCObject`$_i$ first creates a strategy `aRMCAlgo`$_i$, and then invokes operations `rmcast()` on it, with the arguments it just received. Strategy `aRMCAlgo`$_i$ builds message `m'`, containing both `m` and `destSet`. It then issues a reliable point-to-point communication with each object referenced in `destSet`; in order to do this, strategy `aRMCAlgo`$_i$ relies on inherited service `rSend()` of context `aRMCObject`$_i$. When message `m'` reaches `aRMCObject`$_t$, one of the target objects, operation `rDeliver()` is triggered by the protocol. Operation `rDeliver()` detects that `m'` is a multicast message and forwards it to `aRMCAlgo`$_t$, the strategy in charge of that particular execution of the reliable multicast protocol.

In distributed systems subject to failures, a reliable multicast cannot be achieved by merely issuing a sequence of reliable point-to-point communications to objects referenced in `destSet`. This is due to the *agreement* property of the reliable multicast primitive, which requires that either all correct objects in `destSet` or none[4] receive message `m` [3]. So, if

[4]This property is some kind of *atomicity* property.



(a) Context and strategy

(b) Strategy as protocol layer

Figure 3: Strategy design pattern in Bast

message `m'` has not been already retransmitted, the strategy re-issues a reliable point-to-point communication with each object referenced in `destSet` (extracted from `m'`), and then invokes `rmDeliver()` on its context `aRMCObject`$_t$, passing it message `m` (also extracted from `m'`). It is this systematic retransmission scheme that insures the agreement property of the reliable multicast.

Using strategies recursively

Using the Strategy pattern for protocol 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 design pattern, but this is transparently managed by inherited operations of π`Object`. In that sense, Bast enables to use 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 `CSSObject`, a protocol class which provides operations `propose()` and `decide()` allowing to solve the consensus problem presented in section 2. On this figure, the gray oval is context class `CSSObject`, while inner white circles are various π`Algo` strategy classes (π being different protocols). For strategy classes π`Algo`, arrows show the connections between provided services (top) and required services (bottom) of each strategy class. For context class `CSSObject`, provided operations are grouped into services on the application layer side (top). Each strategy class pictured in figure 4 is managed by the corresponding context in the inheritance 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; in particular it can be customized to build abstractions that support transactional systems and group-oriented systems [13]. Elsewhere [8], we present how DTM related protocol classes can be customized to solve the atomic commitment problem, which is central to transactional systems, and the total order multicast, which is central to group-oriented systems.

Overview of the DTM protocol. The protocol starts by the invocation of operation `dtmcast()` on `aDTMObject`$_i$, the initiator object, passing it a message `m` and a set of 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.

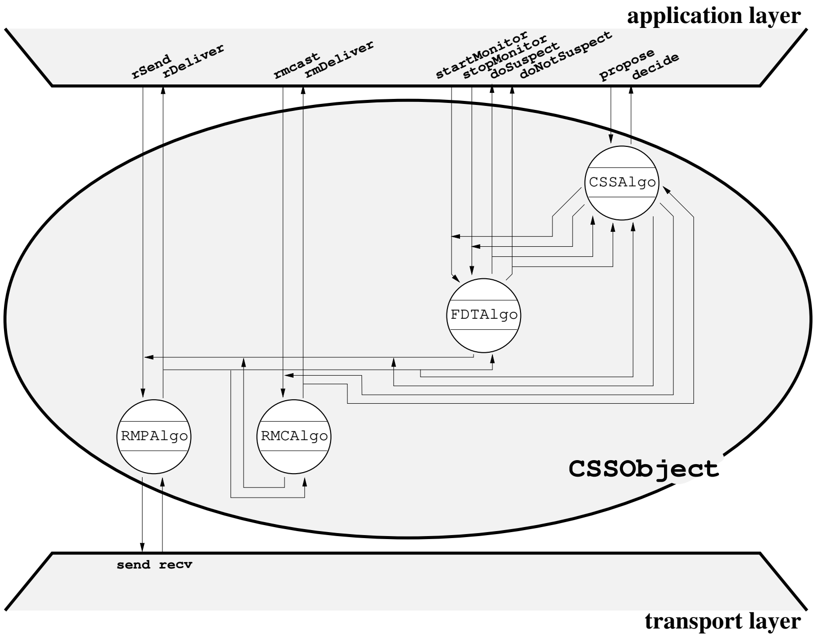


Figure 4: Recursive use of the Strategy pattern

Methodology based on the Strategy pattern. A five steps methodology guides programmers in extending the Bast framework using the Strategy design 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: failure detections, consensus, reliable point-to-point and reliable multicast communications (see [13] 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 inherited services of class `DTMObject`.

4 Related work

The Asx framework [16] 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+ [14], various patterns can be used to build low-level protocol stacks, such as ATM signaling protocols or TCP/IP. This framework basically offers two kinds of objects, *conduits* and *information chunks*, that can be assembled in order to create protocol layers and protocol stacks. The use of design patterns here is motivated by the fact that traditional layered architecture do not allow code reuse across layers, which is precisely what

CONDUITS+ aims at.

There is no such thing as protocol classes in either of the above frameworks. Both ASX and CONDUITS+ can be seen as toolboxes containing sets of reusable components and associated design patterns: in ASX components are wrappers, while in CONDUITS+ they are basic elements that helps building protocol layers. As far as protocol composition goes, ASX provides no such facility. With CONDUIT+ on the other hand, protocols can be composed, 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. However, since the CONDUIT+ framework does not allow to manipulate protocol layers as objects, but only *pieces* of protocol layers, it does not really promote protocol composition the way BAST does it. 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. Another difference is that CONDUIT+ is a pure *black-box* framework, while BAST combines features of both *black-box* and *white-box* frameworks[5].

5 Concluding remarks

In this paper, we introduced the notion of protocol class, and presented how it helps structuring reliable distributed systems through protocol composition. We focused on how the Strategy design pattern enables to overcome the limitations of inheritance, when trying to compose protocols in a flexible manner. In the design pattern community, both the ASX [16] and the CONDUITS+ [14] frameworks are based on reusable object-oriented components and on associated design patterns. Because of the nature of components they provide, ASX does not support protocol composition, whereas CONDUITS+ does. However, in the CONDUIT+ framework, it is not possible to directly manipulate protocol layers as classes of objects. Furthermore, the notion of protocol class is not supported in either of those frameworks. As far as we know, BAST is the only framework relying on protocol classes that address reliability issues. So, BAST is our contribution to the design of well-structured reliable distributed systems.

Future of the BAST framework Our first prototype of the BAST framework is implemented in Smalltalk and 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.

[5] 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] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III Proceedings (DCCA-3)*, pages 197–207. IFIP Transactions, 1993. Elsevier.

[2] M. Aksit, K. Wakita, J. Bosh, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer Verlag, 1993.

[3] 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.

[4] J.O. Coplien and D.C. Schmidt, editors. *Pattern Languages of Program Design*. Addison Wesley, 1995.

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

[6] 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.

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

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

[9] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Implementation of the GARF replicated object plateform. *Distributed Systems Engineering Journal*, 2:14–27, 1995.

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

[11] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, Inc., October 1995.

[12] Object Management Group and X/Open. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1990. Document No. 91.12.1 (Revision 1.1).

[13] 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.

[14] 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.

[15] M.C. Little and S. K. Shrivastava. Object replication in Arjuna. Technical report, University of Newcastle, 1994.

[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] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'92)*, pages 414–434. ACM Press, October 1992. Special Issue of Sigplan Notices.