MODELING PROTOCOLS AS OBJECTS FOR STRUCTURING RELIABLE DISTRIBUTED SYSTEMS

Benoît Garbinato Pascal Felber Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation
Ecole Polytechnique Fédérale de Lausanne (Suisse)
bast@lse.epfl.ch

Keywords: distributed systems, protocol modeling, fault-tolerance, software engineering.

ABSTRACT

When building distributed applications, one has to deal with many complex issues, some of which can be modeled as interacting distributed protocols. Modern operating systems already offer communication facilities such as sockets or RPC-like abstractions, but they have proven to be too low-level or limited in scope. Furthermore, they lack the ability to seamlessly cope with failures. In this paper, we claim that modeling protocols as reusable objects is a good way of structuring distributed systems, since robust protocol implementations and well-tested code reuse are key features in this context. We show how protocol objects can help to build new distributed protocols by composing existing ones, and that they add flexibility to distributed systems. We also give an overview of the design and implementation of BAST*, our extensible library of protocol objects.*

1 INTRODUCTION

This paper introduces the concept of protocol objects and how it can be used to model distributed systems. In particular, we focus on flexible protocol composition through the use of protocol object classes. The paper also describes BAST, an extensible library of protocol classes that is aimed at helping system programmers in building reliable distributed applications. In two companion papers, we have presented (1) how agreement protocol classes can be customized to solve the atomic commitment and the total order multicast (Garbinato et al., 1996b), and (2) how the Strategy design pattern has been extensively used to build BAST (Garbinato et al., 1996a).

1.1 Support For Distributed Systems

The CORBA standard (Object Management Group and X/Open, 1990) is a good example of how objects have deeply affected the way people model and implement distributed systems. In CORBA, specialized abstractions are defined in order to support distributed objects, i.e., mainly remote invocation facilities through well-defined interfaces. On the other hand, although not object-based, most modern operating systems have offered communication abstractions, such as sockets and remote procedure calls (RPC), for quite a long time now. In fact, the core CORBA framework can be seen to merely offer RPC-like invocation facilities between objects. However, sockets are known to be very low-level tools, while RPC have limited scope as far as complex protocols are concerned. Besides, those abstractions are very primitive when it comes to deal with reliability issues, i.e., when one wants to seamlessly cope with failures. Many systems do provide facilities to achieve reliability (Birman and Renesse, 1993; van Renesse and Birman, 1995; Mishra et al., 1993; Shrivastava et al., 1991), but they are closely bound to either the group model or to the transaction model.

1.2 Why Model Protocols as Objects?

Building distributed systems is challenging, because many complex problems have to be dealt with, e.g., remote communications, failures detections, duplication management, etc. Each of these problems can actually be seen as implying the need for a particular distributed protocol, and there are many. Programmers have to make their way across this protocol "jungle", in order to choose the right set of solutions for their problems. In this context, we believe that bare communication facilities are not enough, and that the group model or the transaction model are too restrictive (e.g., what happens if one needs both transactions and groups?). We claim that there is the need for more basic, yet powerful and flexible abstractions. So, we propose to model protocols as objects and to make them the structuring elements of distributed systems. We pretend that protocols as objects can help to factorize distributed algorithms, and to build complex distributed protocols through composition of more basic ones.

1.3 Paper Overview

Next section introduces the notion of *protocol object* and how it can be used as structuring component to model protocol relationships; protocol composition, which is achieved through the concept of *algorithm object*, is also presented. The BAST protocol class library is then described in section 3; its architecture based on layered services, together with some implementation issues of our first prototype, are discussed there. Section 4 compares our approach with other research works in the literature, and section 5 summarizes the contributions of the paper to the modeling of distributed systems.

2 PROTOCOLS AS OBJECTS

The *distributed system* we consider is composed of *distributed objects* that communicate by message passing, i.e., any pair of distributed objects have the means to remotely designate each other, and to send messages to each other. Message passing communications are point-to-point and can have different reliability characteristics, e.g., unreliable, best-effort, reliable, fifo, etc. Distinct distributed objects are located *a priori* on different network nodes, which imply different address spaces, and faulty objects fail by crashing.

2.1 Distributed Protocols

We define *distributed protocol* π as a set of interactions between distributed objects that are aiming at solving some *problem* π. With such a broad definition, any of the message passing communications mentioned above can be seen as a distributed protocol: one has simply to choose an adequate specification for problem π. However, our definition covers many other distributed protocols than mere point-to-point communications, e.g., reliable multicast, failure detection, distributed consensus, etc.

In order to seamlessly cope with failures, a distributed application must be capable of hiding failures, i.e., of enforcing its *liveness*, and capable of preventing failures from putting it in a inconsistent state, i.e., of enforcing its *safety* (Ricciardi et al., 1993). To achieve this, distributed applications are based on reliable distributed protocols. The implementation of such protocols implies that one has to deal with many other underlying protocols, which results in complex dependency relationships. A good example of such a complexity is the distributed consensus protocol: it is based on reliable message passing communications (point-to-point), on failure detections and on reliable multicast communications. In turn, the reliable multicast protocol relies on reliable message passing communications, possibly on failure detections. Figure 1 (a) presents some protocol dependencies.

2.2 Protocol Objects

In this context, programmers have to choose the right protocol depending on their need, and when more than one is necessary, comes the problem of protocol interactions which further complexifies their task. So, we believe that protocols should be basic components of distributed systems, and they should be manipulated as *classes of objects*.

We define a *protocol object* of some class π`Object` as a distributed object capable of participating to protocol π; class π`Object` is referred to as a *protocol class*. In this paper, π`Object` designates any class of protocol objects, while aπ`Object` designates some protocol object of class π`Object`. Protocol classes are organized into a single inheritance hierarchy and abstract superclass `Protobject` is the root of this hierarchy. The latter is presented in figure 1 (b) and follows protocol dependencies pictured in figure 1 (a). Each π`Object` class implements only one protocol and provides operations that act as entry points to the protocol, e.g., class `RMPObject` implements operations `rSend()` and `rDeliver()` allowing to reliably send, respectively receive, a message; operation `rDeliver()` is a callback and is said to be triggered by the protocol. However, protocol objects of some π`Object` class can execute any protocol inherited from π`Object`'s superclasses. Furthermore, protocol objects can concurrently participate to several distributed protocols (identical or different).

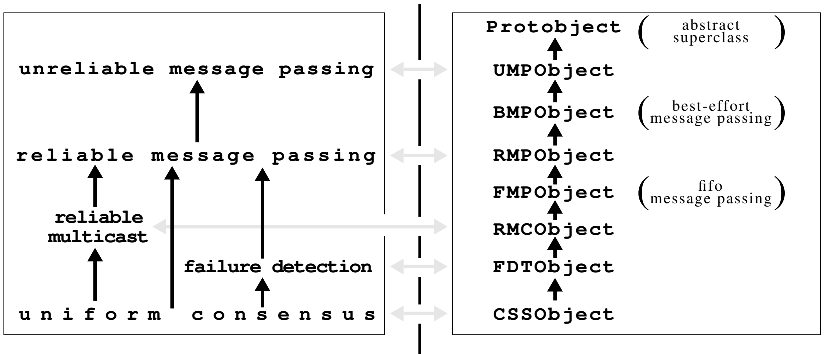
Reliability in distributed systems is closely related to the intrinsic robustness of protocol implementations, and reusing well-tested code is thus a key feature. The concept of protocol objects helps to reuse complex distributed algorithms, e.g., the distributed consensus algorithm described by Chandra and Toueg (Chandra and Toueg, 1994) is implemented, once and for all, in a reusable protocol class. Protocol objects also help to understand relationships between various distributed protocols, and making them work together is then made easier. Subclassing allows to customize and/or optimize existing protocols, and to create new ones with minimal efforts.

Generic Protocols. We believe that arguments passed to protocol operations should be as unconstraint as possible. For example, protocol objects of class `RMPObject` are allowed to use operation `rSend()` to reliably send *any* object that is not a protocol object. So, in our approach, distributed protocols are said to be *generic*. Here, genericity also means that callback operations, such as `rDeliver()`, have to be redefined in order to be used by applications.

2.3 Interacting Protocol Layers

With our approach, protocol dependencies have to be managed not only during the design phase (between protocol classes), but also at runtime (between protocol objects); the fact that protocol objects can concurrently execute several protocols is the reason for this. In this context, another way to look at protocol dependencies is to view protocols as various layers which depend on each other. So, in order to compose protocols one has to answer the question "*When and how are protocol layers assembled, and how do they cooperate?*". We now present under which circumstances (*when*) various protocol layers are assembled at runtime, through the example of the distributed consensus protocol. In Section 3, we present in further details a full execution of the distributed consensus protocol (in particular *how* protocol layers are actually assembled).

Assembling Distributed Consensus. The distributed consensus problem can be defined as follows: all distributed objects in some set σ propose an initial value and must reach agreement on one of the proposed values (the decision) (Fischer, 1983). It can also be specified in terms of two primitives, *propose()* and *decide()*, which mark the beginning and the termination of the protocol respectively (Chandra and Toueg, 1994). Class `CSSObject` implements protocol objects capable of solving the distributed

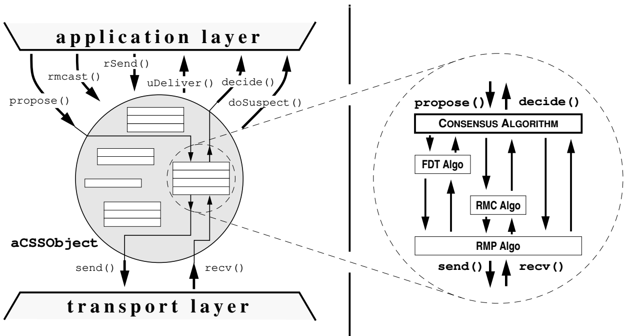


(a) protocol dependencies (b) protocol class hierarchy

Figure 1: Protocols and Protocol Classes

consensus problem, and defines operations `propose()` and `decide()` to do so. Note that protocols objects of that class are not only capable of participating to distributed consensus protocols, but also to any protocol inherited by class `CSSObject` (see figure 1 (b)); for example, such protocol objects can perform reliable message passing communications and failure detections.

Figure 2 (a) presents a snapshot of protocol object `aCSSObject`, taken at runtime. In this figure, `aCSSObject` is concurrently participating to five different protocols; on top, we pictured the application layer that is using it, while at the bottom, one can see `aCSSObject` issuing low-level calls to the transport layer. Each protocol execution launched by the application is managed by an independent protocol stack, and each layer within a stack is in charge of one of the lower-level protocols on which the protocol launched by the application is relying. Whenever the application invokes operation `propose()`, a new execution of the distributed consensus protocol is started. This causes the `aCSSObject` to assemble various protocol layers (at runtime) into protocol stack pictured in figure 2 (b).



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

Figure 2: Protocol Objects and Protocol Layers

2.4 Composing Protocols

Our first attempt to model protocol interactions was based only on strictly vertically layered protocol stacks, but we were forced to face the fact that such a model does not always make sense in terms of protocol composition, as suggested by figure 2 (b). In the end, only the *encapsulation* benefit of the layered approach remained, while its strictly vertical nature was discarded. Our conclusion here concurs the work done by Hüni and al. on their framework for network protocols (Hüni et al., 1995).

Limitations of Inheritance. When trying to compose protocols, we also realized that inheritance alone is not sufficient because it does not offer enough flexibility. For example, it does not allow to use a new protocol algorithm in whatever protocol object we want. Besides, inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*. These limitations are mainly due to the fact that inheritance forces code reuse through subclassing.

Our solution to overcome this lack in flexibility is to *objectify* protocol algorithms, i.e., to make separate objects out of them. As a consequence, any π`Object` protocol class becomes independent of the algorithm supporting protocol π, which makes protocol composition very flexible. In a companion paper (Garbinato et al., 1996a), we present in details how objectifying protocol algorithms comes down to recursively apply the *Strategy design pattern** described by Gamma and al. (Gamma et al., 1995).

Algorithm Objects. We define an *algorithm object* of some class π`Algo` as an object implementing some distributed algorithm for solving problem π; any protocol algorithm class is derived from abstract class `ProtoAlgo`. An algorithm object always executes *within* a protocol object, and the two are thus tightly coupled. Furthermore, the application layer only deals with protocol objects, i.e., it knows nothing about algorithm objects.

The relationship between protocol objects and algorithm objects is pictured in figure 3. Protocol object `a`π`Object` offers two kinds of operations to the application layer: operations it inherits from its superclasses and operations that are specific to protocol π. As far as protocol π is concerned, the algorithm implementing it is part of object `a`π`Algo`'s code, while `a`π`Object` merely uses the corresponding operations provided by `a`π`Algo`.

*Design patterns allow to capture recurring object-oriented solutions.

In turn, operations required by aπAlgo to run protocol π are based on the inherited operations of aπObject (remember that most protocols depend on others); this is precisely where protocol composition occurs. As a consequence, when some operation related to protocol π is invoked on aπObject, its execution is delegated to aπAlgo. Symmetrically, when some operation on which protocol π relies is needed by aπAlgo, the latter invokes the corresponding (inherited) operation on aπObject.

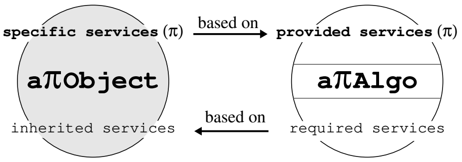


Figure 3: Protocol and Algorithm Objects

Each instance of class πAlgo represents one execution of protocol π's algorithm implemented by that class. At runtime, aπObject maintains a table of all algorithm objects concurrently executing within him, and all messages are tagged in order to identify in which execution of what protocol they have to be delivered. 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.

Flexible Protocol Composition. The interaction scheme presented in figure 3 requires aπObject to be capable of executing all lower-level protocols on which protocol π is based. Distributed protocol programmers are then forced to clearly specify what underlying protocols are required to implement protocol π. Only then, programmers can choose which protocol class to subclass in order to implement πObject. This approach also makes it possible to change any protocol algorithm of any protocol class: for example, one could optimize the failure detection algorithm in CSSObject's superclass and in one of its subclasses, while leaving it unchanged in class CSSObject. Algorithms could even be chosen at runtime, according to some criteria computed during the execution. In other words, dependencies between protocols remain only at the specification level (i.e., through subclass relationships depicted in figure 1 (b)), whereas the implementations (i.e., algorithm objects) can vary. Compared to the traditional protocol stack approach, our model is easier to use yet more flexible. It offers ready-to-use components (protocol objects) to application programmers, and provides system programmers with an open and non-strictly vertically layered model.

Another advantage is that system programmers can strictly focus on protocol π and on the interaction between class πObject and class πAlgo: 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 πObject. Such operations may also be based on one or more algorithm objects, but this is automatically managed by superclasses of πObject.

3 THE BAST LIBRARY

The concept of protocol object is the cornerstone of BAST, an extensible library of protocol classes aimed at facilitating the task of programmers, more specifically those who have to deal with distributed systems subject to crash failures. The BAST library is part of *Phoenix*, an on-going research project in the domain of reliable distributed systems, at the Operating Systems Laboratory of the Swiss Federal Institute of Technology in Lausanne. BAST was named after the cat-goddess of the Egyptian mythology[†].

3.1 Extensible Architecture

The BAST protocol class library has a layered architecture, each layer being composed of a set of classes that implement one or more related protocols. At the bottom, the communication service relies on the transport layer, i.e., a layer capable of performing remote communications between different network nodes; at the top, the application layer is using protocol objects. As already mentioned, our modeling of protocol interactions is not strictly vertically layered, so distributed applications can use any BAST's service, i.e., any corresponding class of protocol objects.

An important feature of BAST is its extensibility, which allows to build new protocol classes from existing ones. With this respect, figure 1 (b) only gives a partial view of BAST's current status: several other classes have been added, that implement various multicast and broadcast protocols, some of which totally ordered, as well as atomic commitment and distributed locking protocols. All those protocols are actually based on a generic agreement protocol, the dynamic-terminating-multicast (DTM), which has been shown to be very useful to build various protocols where agreement is essential (Guerraoui and Schiper, 1995). In particular, we have presented elsewhere (Garbinato et al., 1996b) how DTM protocol classes can be customized to solve the atomic commitment and the total order multicast, which are central problems to transactional models and to group-oriented models respectively.

3.2 Current Status

Our first implementation of BAST is fully operational and was written in Smalltalk (Goldberg and Robson, 1983). More specifically, the development took place on top of VisualWorks 2.0, the commercial Smalltalk environment by ParcPlace Systems, Inc., on a network of Sun SPARCstations running operating system Solaris 2.4; the communication facilities of the latter were used to implement BAST's transport layer. A C++ version of BAST is currently being worked on, in the context of the *Opendreams* project (ESPRIT n$^{\circ}$29843), which aims at designing a *reliable* CORBA compliant platform (Felber et al., 1996). However, we still intend to use our Smalltalk prototype as reference version for further developments, and to add more and more protocol classes to it in order to further test our approach. We also consider porting BAST to Java (Gosling and McGilton, 1995), since this new programming language shares many common features with Smalltalk, and seems to be a very promising tool as far as distributed processing goes.

[†] Cats are said to resist to a fair number of “crashes” ( ).

3.3 Basic Services

`Protobject` and `ProtobjectRef` are two fundamental classes in BAST: `Protobject` is the root of the protocol class hierarchy, i.e., any protocol object (no matter its actual class) can be seen as an instance of that class, while instances of `ProtobjectRef` designate remote protocol objects and are passed as arguments to communication operations. However, those two classes do not provide any support for remote communications; in fact, they are associated to no protocol whatsoever. Class `ProtoAlgo` is the superclass of any protocol algorithm class used by protocol objects, but remember that algorithm objects are completely invisible to application programmers using BAST. As already mentioned, the BAST protocol class library is organized into layered services, each being composed of one or more protocol classes derived from `Protobject`. We now briefly present the basic services provided by BAST.

Communication Service. Since Smalltalk offers no support for distributed objects, we had to implement the foundation for remote object communications. This service is the base of all other distributed protocols and defines five protocol classes: `UMPObject`, `BMPObject`, `RMPObject`, `FMPObject`, and `RMCObject`. The first four classes all implement *point-to-point* communications through message passing, but they differ in their delivery guarantees, i.e., they provide unreliable, best-effort, reliable and fifo point-to-point message passing respectively. With these four classes, sending messages is achieved by invoking some x`send()` operation, where $x \in \{\mathtt{u}, \mathtt{b}, \mathtt{r}, \mathtt{f}\}$ corresponds to one of the above mentioned delivery guarantees, and receiving in triggered through a callback to the corresponding x`deliver()` operation. Protocol class `RMCObject` implements reliable *multicast* (point-to-multiple-points) communications, through operation `rmcast()` for sending and callback operation `rmDeliver()` for receiving. Each class but `UMPObject` also offers callback operation y`done()`, where $y \in \{\mathtt{b}, \mathtt{r}, \mathtt{f}, \mathtt{rm}\}$, which indicates that the communication was successful.

All point-to-point communications are performed by *private* class `TransportLayer`, which represents the transport layer on each network node. In our current Smalltalk prototype, a node corresponds to a *Smalltalk virtual machine* (ST-VM) in execution and callbacks are triggered by class `TransportLayer`. In order to implement communications between ST-VMs, class `TransportLayer` relies on a low-level layer written in C: *the Reliable Communication Layer* (*RCL*), also developed as part of the *Phoenix* project. RCL is based on UDP sockets and extends them with the delivery guarantees mentioned above.

Since the actual protocol algorithms for point-to-point communications are implemented in RCL, none of the point-to-point message passing classes needs to rely on algorithm objects. This approach has the advantage to improve communication performance, since low-level management, such as message packing and retransmitting, duplicated messages filtering, etc., is written in C. Class `TransportLayer` is only responsible for marshaling and unmarshaling messages sent through RCL, and for dispatching incoming messages to local protocol objects (by triggering the right delivery callback operation). However, RCL offers no support for reliable *multicast* communications, so class `RMCObject` is the first protocol class in BAST's inheritance hierarchy to rely on a protocol algorithm class, namely `RMCAlgo`. The latter implements a naive algorithm based on systematic retransmissions when messages reach their target objects for the first time; details can be found in (Chandra and Toueg, 1994; Garbinato et al., 1996a).

Failure Detection Service. Instances of class `FDTObject` are protocol objects capable of executing *the failure detection protocol*: they manage a private set containing the `ProtobjectRefs` of every remote protocol object currently suspected to be faulty (i.e., to have crashed). Class `FDTObject` implements operations `startMonitoring()` and `stopMonitoring()`, used to start and stop "pinging" protocol objects respectively, as well as callback operations `doSuspect()` and `doNotSuspect()`, which are triggered whenever a protocol object being currently monitored should, or should no more, be suspected respectively. Those two last operations are responsible for updating the private suspicions set managed by each `FDTObject` instance.

Our current prototype makes the assumption that protocol objects on the same network node do not fail *independently*, i.e., we only consider failures of ST-VMs, not of individual objects. The actual monitoring of remote ST-VMs is performed by RCL, on behalf of class `TransportLayer`. RCL implements a failure detector of type $\diamond S$, which is said to be *unreliable* because it can make false suspicions[‡]; see (Chandra and Toueg, 1994) for a complete discussion on failure detector classification. Class `TransportLayer` is merely responsible for dispatching suspicions and non-suspicions to all concerned local protocol objects. This is achieved as follows: whenever a message is received on node A from some protocol object located on node B, all objects on node A that are suspecting some object located on node B are invoked through callback `doNotSuspect()`; symmetrically, whenever some protocol object on node B is no more reachable by some object on node A, all objects on node A that interested in some object located on node B are invoked through callback `doSuspect()`. Similarly to protocol classes supporting point-to-point message passing, class `FDTObject` does not rely on an algorithm class, since failure detection algorithms are directly implemented in the underlying transport layer (RCL).

Distributed Consensus Service. Support for solving the agreement problem, which is central to many reliable distributed protocols, is provided by class `CSSObject`. This protocol class defines operation `propose()`, which starts the execution of the distributed consensus, and callback operation `decide()`, which is triggered by the protocol when an agreement has been reached.

The actual distributed consensus protocol is implemented in algorithm class `CSSAlgo`, which implements the $\diamond S$-consensus protocol described by Chandra and Toueg (Chandra and Toueg, 1994). This algorithm enables protocol objects to reach agreement in asynchronous systems augmented with failure detectors of type $\diamond S$. Objects participating in such distributed consensus algorithm must be capable of suspecting each other; class `CSSObject` is

[‡]This type of failure detectors have been proved to be sufficient to solve the distributed consensus problem in asynchronous systems.

therefore a subclass of `FDTObject`, in order to offer failure detection operations to its algorithm objects (instances of class `CSSAlgo`). The algorithm also requires protocol objects to be able to perform reliable point-to-point and multicast communications, so class `CSSObject` is also an indirect subclass of both `RMPObject` and `RMCObject`.

3.4 Distributed Consensus in Execution

We now present the complete execution of the distributed consensus by an instance of protocol class `CSSObject`, in order to show how the latter manages several algorithm objects. It should also help to understand how such algorithm objects are transparently managed by the various protocol class levels, through inherited operations. In the example, only one execution of the distributed consensus is considered (no other concurrent protocol is running).

As long as no agreement is reached, distributed consensus algorithm `aCSSAlgo` only needs to perform reliable point-to-point message passing and failure detection; corresponding operations are available because `aCSSObject` inherits from classes `RMPObject` and `FDTObject`. We have seen above that RCL provides support for this, so only algorithm object `aCSSAlgo` is involved at that stage. Upon taking the decision, `aCSSAlgo` issues a reliable multicast communication to all objects participating to the distributed consensus protocol. Since RCL provides no support for reliable multicast, `aCSSAlgo` transparently uses algorithm object `aRMCAlgo`, through operations `rmcast()` and `rmdeliver()` that `aCSSObject` inherits from protocol class `RMCObject`.

4 RELATED WORK

4.1 Representing Distributed Systems

CHOICES (Campbell et al., 1993) is an attempt to model traditional components of distributed systems through C++ classes with well-defined interfaces, e.g., *Process*, *Domain*, *Disk*, etc. The basic idea is to enable programmers to customize distributed systems, through inheritance, in order to meet particular needs. In a similar way, BETA (Brandt and Madsen, 1993) defines a set of "mandatory" abstractions to allow distributed object execution, such as *Ensemble*, representing network nodes, and *Shell*, representing self-contained program modules. Again, the goal is to make it possible to customize distributed systems in order to meet specific application needs. All those systems focus on distributed object executions and define several abstractions, whereas our approach is based on only two concepts: protocols as objects and their composition through protocol algorithm objects.

4.2 Representing Communication Protocols

Systems like ISIS (Birman and Renesse, 1993), HORUS (van Renesse and Birman, 1995) and CONSUL (Mishra et al., 1993) all provide reliable distributed protocols, but groups are the elemental distributed addressing facilities. The ARJUNA system (Shrivastava et al., 1991) allows to deal with failures by providing communications based on transactions. Because these approaches strongly couples reliability issues either to groups or to transactions, they limit the scope of such systems and hinders extensibility. STREAMS (Ritchie, 1984) and the *x*-Kernel (Peterson et al., 1989) also offer libraries of communication protocols, but they do not deal with reliability issues at all. Besides, no framework for protocol composition is clearly given. As presented in this paper, protocol objects are BAST's distributed addressing entities on top of which groups communication protocols can be implemented, e.g., reliable multicast communications, as well as transactions. Furthermore, the interaction scheme between protocol objects and algorithm objects provides a flexible framework for composing protocols.

The ASX framework (Schmidt, 1994) introduces a set of C++ collaborating components, known as *wrappers*, in order to facilitate building reusable communication infrastructures. Those components are aimed at performing communication-related tasks, such as event demultiplexing, connection establishment, routing, etc. As architectural blueprints, various design patterns are provided, e.g., the *Reactor* and the *Acceptor* patterns, which help building communication infrastructures in a modular way. The CONDUITS+ framework (Hüni et al., 1995) offers components and design patterns to build low-level protocol stacks, e.g., ATM signaling or TCP/IP protocols. Basically, two kinds of objects are provided: *conduits* and *information chunks*, that can be assembled in order to create protocol layers and protocol stacks. This approach is motivated by the fact that traditional layered architecture do not make it possible to reuse code across layers, which is what CONDUITS+ allows.

However, none of the above frameworks provide the unifying concept of protocol object: ASX and CONDUITS+ can rather be seen as toolboxes of reusable components. Furthermore, ASX provides no facility for protocol composition, whereas CONDUIT+ does but at a lower-level than BAST (by assembling elementary blocks to build protocol layers). Protocols are further decomposed in CONDUIT+ than with our approach: conduits and information chunks are finer grain objects than algorithm objects. Such fine grain objects are powerful tools to reuse code in protocol implementations, but they also require more work to achieve protocol composition.

5 CONCLUSION

In this paper, we introduced the notion of protocol object, and presented how it can help to model distributed systems by allowing to flexibly compose protocols through the use of algorithm objects. We claimed that protocol composition is appropriate to structure distributed systems, because this task involves dealing with many complex protocols. Other previous research works have already tackled the problem of distributed system design, either by providing specialized abstractions or by providing libraries of communication protocols, but none of them promotes protocol composition through protocol and algorithm objects, as BAST does it. We see our approach, which is implemented in the BAST protocol class library, as a contribution to the modeling of well-structured distributed systems.

REFERENCES

Birman, K. and Renesse, R. V. (1993). *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press.

Brandt, S. and Madsen, O. (1993). Object-oriented distributed programming in Beta. In *Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 185–212. Springer Verlag.

Campbell, R., Islam, N., Ralia, D., and Madany, P. (1993). Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126.

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

Felber, P., Garbinato, B., and Guerraoui, R. (1996). The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems (SRDS'96)*.

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

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

Garbinato, B., Felber, P., and Guerraoui, R. (1996a). Composing reliable distributed protocols using the Strategy pattern. Technical report, Operating System Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology.

Garbinato, B., Felber, P., and Guerraoui, R. (1996b). 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.

Goldberg, A. and Robson, A. (1983). *SMALLTALK-80: The Language and its Implementation*. Addison Wesley.

Gosling, J. and McGilton, H. (1995). The Java language environment: A white paper. Technical report, Sun Microsystems, Inc.

Guerraoui, R. and Schiper, A. (1995). 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.

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

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

Object Management Group and X/Open (1990). *The Common Object Request Broker: Architecture and Specification*. Object Management Group (OMG). Revision 1.1.

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

Ricciardi, A., Schiper, A., and Birman, K. (1993). Understanding partitions and the "no partition" assumption. In *Proceedings of the IEEE 4th Workshop on Future Trends of Distributed Systems (FTDCS-93)*.

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

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

Shrivastava, S., Dixon, G., and Parrington, G. (1991). An overview of the Arjuna distributed programming system. *IEEE Software*.

van Renesse, R. and Birman, K. (1995). Protocol composition in Horus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'95)*.

ABOUT THE AUTHORS

All three authors are computer science engineers and work as researchers at the Operating Systems Laboratory, in the Swiss Federal Institute of Technology of Lausanne. Their interests include distributed systems, fault-tolerance, as well as object-oriented technologies. In this context, authors are also deeply involved in the CORBA research stream.