BAST
A Framework for Reliable Distributed Computing

Benoît Garbinato Rachid Guerraoui

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

Abstract

Although useful in the development of distributed systems, current reliable distributed environments, such as group communication toolkits (e.g., ISIS, TOTEM, PHOENIX) or transactional monitors (e.g., ENCINA, TUXEDO), are hardly extensible or customisable for specific application needs. The limitations of those systems are mainly due to their underlying distributed protocols, which are usually designed and implemented in an *ad hoc* manner.

This paper presents BAST, an object-oriented framework for building fault-tolerant distributed applications. We also show how BAST is used to implement distributed protocols, by providing centralised and distributed *design patterns*. In particular, we present how we built atomic commitment and atomic multicast protocols, which provide adequate support for a wide range of fault-tolerant distributed applications. We detail customisation facilities of BAST and discuss our design in the light of different alternatives. Finally, we present BAST implementations in Smalltalk and Java, and we point out some optimisation issues.

1 Introduction

BAST[1] is a framework of reliable distributed protocols. In comparison with similar protocol libraries such as Horus [35], BAST achieves one further step towards modular and flexible protocol composition.

1.1 Reliable Distributed Systems

"A distributed system is one that stops you from getting any work done when a machine you've never even heard of crashes." Leslie Lamport *in* [26].

Reliable distributed systems are challenging to build, because programmers have to deal with many complex issues, e.g., reliable communication, failure detection, replication management, transactions management, etc. Each of these issues actually corresponds to a distributed problem to solve, and programmers have to face a real "jungle" of protocols (Figure 1). Depending on the programmer's skills,

[1] In the ancient Egypt, *Bast*, also known as *Bastet*, was a cat-goddess worshipped in the delta city of Bubastis. Since cats are said to benefit from several lives, they are in some sense tolerant to a fair number of "crashes". So, we thought the name of the protectress of cats would be an adequate one for our reliable distributed framework.

the latter can either choose the right protocol for the right application from some library, or build new protocols.

The intrinsic robustness of protocol implementations is an important issue for achieving fault tolerance and reusing well-tested code is a key feature. Since relationships between protocols can be quite complex, there is the need to structure this complexity in order to reuse existing code. In this context, we believe that protocols should be basic structuring components of distributed systems. In the BAST framework, distributed protocols are manipulated as *classes of objects*. Such classes make it very easy to use, as well as to extend and/or customise, abstractions provided by distributed environments based on BAST. Manipulating protocols as classes allows to structure fault-tolerant distributed applications according to their needs in protocols, and simplifies the building of new complex protocols, when combined with adequate design patterns[2] [10].

Terminology Used in this Paper

In the remainder of this paper, we choose to categorise programmers according to their skills and we give them nicknames. *Bob* will be our emblematic *application programmer*, who merely wants to use BAST to develop reliable distributed applications and knows nothing about protocol design, while *Alice* will be our emblematic *protocol programmer* who wants to extend BAST with new protocol classes. We also distinguish the terms *distributed applications* and *distributed environments*, when talking about what Bob and Alice are producing respectively. We use *distributed systems* when a more general term is needed. By *reliability* or *fault tolerance*, we mean the capability of hiding failures, i.e., of enforcing the *liveness* of the system, as well as the capability of preventing failures from putting it in a inconsistent state, i.e., of enforcing the *safety* of the system.

1.2 Fault Tolerance in BAST

A *fault-tolerant* distributed environment can be described as one that provides abstractions capable of hiding failures to its users (at least to some extent), and of preventing failures from putting it in a inconsistent state. In this context, two main paradigms for building reliable distributed applications have emerged over the years: the *transaction paradigm*, originated from the database community, and the *group paradigm*, originated from the distributed systems community. Each one of those two paradigms is tailored to solve a particular set of problems.

Transactional Protocols

Many distributed environments provide the *transaction paradigm* [24] as the main building block for programming reliable applications. This concept has proven to be very useful for distributed database-like applications. However, the *ACID*[3] properties of the original transaction model are too strong for several applications. For example, the *Isolation* property is too strong for cooperative work applications, whereas the *All-or-nothing* property is too strong for applications dealing with replicated data. This is partly due to the fact that underlying agreement protocols are designed and implemented in an *ad hoc* manner and cannot be modified. The rigidity of the original transaction model has lead many authors to explore the design of more flexible transactional models, e.g., nested transactions, but the underlying agreement protocol still cannot be modified.

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

[3] *All-or-nothing*, *Consistency*, *Isolation*, and *Durability*.

Figure 1: Bob and Alice facing the distributed protocol "jungle"

In designing the BAST framework, we have adopted an alternative approach, which consists in providing the basic abstractions required to implement various transaction models, rather than supporting one specific model. These abstractions implement support for reliable total order communications (allowing to build distributed locking), atomic commitment, etc., and are aimed at being used by Alice, not Bob. As we shall see in Section 5, those abstractions are based on a *distributed design pattern* which captures the essence of most agreement protocols.

Group Communication Protocols

Group-oriented toolkits like ISIS [3] or GARF [14] offer reliable communication primitives with various consistency levels, e.g., causal order multicast, total order multicast. These environments are based on the *group paradigm* as fundamental abstraction for reliable distributed programming. The group concept is very helpful to handle replication: a replicated entity (a process in ISIS or an object in GARF) is implemented as a group of replicas. It constitutes a convenient way of addressing replicated logical entities without having to explicitly designate each replica. When a failure occurs, members of a group are notified through a *group membership* protocol, and can act consequently. This is useful, for example, when implementing a primary-backup replication scheme: if the primary replica crashes, the backups replicas are notified through some membership protocol, and can then elect a new primary.

Group membership protocols guarantee that all members of some group g agree on a totally ordered sequence of views $view_1(g), view_2(g), \ldots, view_n(g)$; a view change occurs each time a member joins or leaves group g. Furthermore, multicasts to group g are guaranteed to be totally ordered *with respect to view changes*. Finer ordering criteria *within each view* are generally also available in environments such as ISIS or GARF, e.g., causal or total orderings. Membership protocols are normally based on agreement protocols. However, the strong coupling between the group concept and consistency leads to the inability to sup-

port reliable multicast that involve different replicated entities, i.e., several groups. This limitation makes group-oriented environments unable to seamlessly integrate transaction models. There again, underlying agreement protocols are hidden and, being not accessible, they cannot be customised.

A major characteristic of the BAST framework is that it allows to decouple the group notion from consistency issues: groups are viewed merely as a *logical addressing capability*, while reliable multicast communications are supported by adequate protocol classes based on a distributed agreement pattern. As a consequence, BAST naturally supports reliable multicasts involving different groups of replicas.

1.3 Status of BAST

Our first implementation of BAST was written in Smalltalk. It has been used for teaching reliable distributed systems, as well as for prototyping new fault-tolerant distributed protocols. BAST was later ported to Java [16], and a Java prototype is now also operational. Both prototypes are currently available on the web as public-free distributions, at *http://lsewww.epfl.ch/bast*. Full documentation as well as several companion papers can also be found there.

1.4 Overview of the Paper

Next section gives an overview of BAST from Bob's point of view, i.e., we describe what application programmers have to know to start building reliable distributed applications with BAST. Section 3 gives an example of how Bob could use BAST to actively replicate a server object in some distributed application, in order to make it fault-tolerant. In Section 4, we present Alice's point of view of BAST, i.e., what protocol programmers must know of its internals, mainly based on two design patterns, in order to be able to extend BAST. Then, Section 5 illustrates the various extensibility features of BAST through three concrete examples. Section 6 discusses some design and implementation issues, and presents performance measures. Section 7 gives an overview of existing systems described in the literature, which address common issues with BAST. Finally, Section 8 points out some concluding remarks.

2 Overview of BAST

The BAST framework is designed to help programmers in building reliable distributed systems, and is based on protocols as basic structuring components. It is aimed at providing a extensible set of powerful abstractions, that support the design and implementation of reliable distributed systems. Figure 2a presents an overview of BAST's architecture, based on a fully object-oriented design and implementation. In BAST, various abstraction levels are provided, depending on whether it is Bob or Alice who is using it. At the highest level, all the complexity is hidden to Bob, in ready-to-use components. Such components can be seen as an evolution of the GARF environment, which was aimed at supporting reliable distributed programming in a fairly automated way [14]. In GARF however, reliability issues were dealt with by ISIS, used as a black-box toolkit, so reliable distributed protocols were not accessible. With the BAST framework on the contrary, protocols are directly made available to Alice, who can customise and/or create protocols from existing ones. In this section, we only present what Bob needs to know to be able to use BAST. Extensibility features are left out and will be presented when we adopt the point of view of Alice.

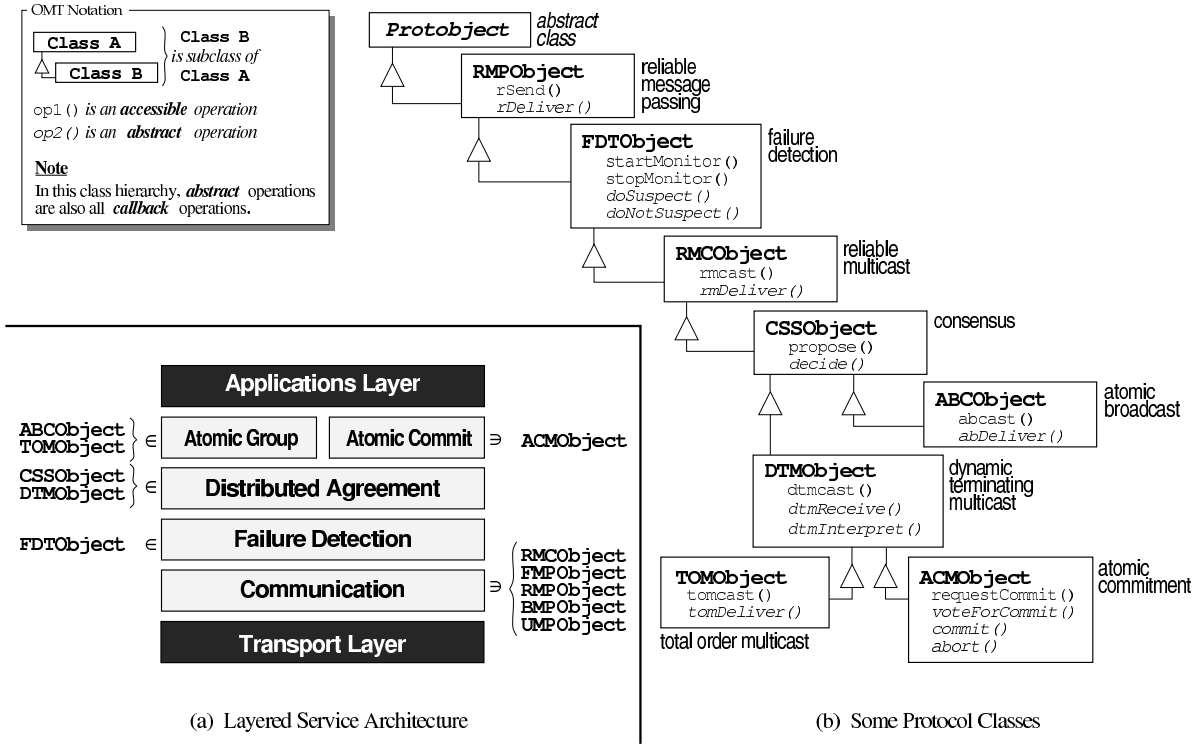


Figure 2: Overview of the BAST framework

2.1 System Model

In the BAST framework, we consider a distributed system composed of objects that have the means to remotely designate each other and to participate in various protocols. Distinct distributed objects reside *a priori* on different nodes, and faulty nodes are supposed to fail independently by crashing. We do not make any assumption on the synchrony of the system, but we use the notion of failure detector instead[4]. We define *distributed protocol* π as a set of coordinated interactions between distributed objects, which aims at solving *problem* π. This implies a very broad view of what distributed protocols are, since it all depends on the specification of problem π. According to this definition, even a trivial unreliable communication between two remote objects is a distributed protocol. We believe however that the generality of this definition helps to make things simpler.

2.2 Ready-to-Use Components

Building fault-tolerant applications is normally a hard task which requires highly skilled protocol programmers, because reliable distributed protocols can be very complex to understand and to implement. To make this task accessible to Bob, we need ready-to-use components supporting adequate distributed protocols. For example, the atomic commitment protocol is essential to any applications involving transactions, while the atomic broadcast protocol is

[4]Failure detectors enable to hide the synchrony of distributed systems and are abstractly defined through reliability properties [5].

the cornerstone of applications based on active replication. In this context, we believe that *protocol classes* are powerful tools for allowing Bob to put complex protocols to work, while remaining easy to understand and to use.

What Are Protocol Classes?

The concept of *protocol class* can be defined as follows: a protocol class π`Object` implements the behaviour of distributed objects capable of executing protocol π, i.e., of solving distributed problem π. Instances of protocol classes are designated as *protocol objects*. We use π`Object` to refer to any protocol class, and aπ`Object` to designate some distributed object of class π`Object`.

Figure 2b presents some ready-to-use protocol classes proposed in BAST. This class diagram uses the standard OMT (Object Modelling Technique) notation [33]; abstract operations are all callback operations. The root of this class hierarchy is abstract superclass `Protobject`. Each π`Object` class provides a set of operations that interface protocol π: these operations act as entry points for applications using protocol π. For example, class `RMPObject`, which implements reliable point-to-point communications, provides operation `rSend()` and callback `rDeliver()` that allow to reliably send, respectively receive, a message.

Service Architecture

The BAST framework is structured into layered services, each being composed of one or more classes that implement related protocols; this is pictured in Figure 2a. Bob is responsible for the application (top layer), while at the bottom, the communication service relies on the transport layer; the latter is responsible for low-level communications between different network nodes and relies on operating system services (not pictured in Figure 2a). Anything in between the application and the transport layers is part of BAST. Our model is not strictly *vertically* layered (although Figure 2a might suggest it), so Bob can use any existing BAST's service. We now give an overview of the classes pictured in Figure 2. Implementation of these classes is discussed in Section 6.

Basic Classes. Bob first has to learn about two classes: `Protobject`, which is the root of the protocol class hierarchy, and `ProtobjectRef`, which enables to designate remote protocol objects. Those two classes provide no support for remote communications or any other protocol: `Protobject` is an abstract class, while `ProtobjectRef` serves only as addressing facility.

Communication Service. This service defines the following classes: `UMPObject`, `BMPObject`, `RMPObject`, `FMPObject`, and `RMCObject`. The first four implement *point-to-point* communications and differ in their delivery guarantees, i.e., they provide unreliable, best-effort, reliable and fifo point-to-point communications respectively. Class `RMCObject` implements reliable *multicast* communications.

Failure Detection Service. Protocol objects of class `FDTObject` are capable of monitoring each other. Each one also manages a private set containing remote references of objects it suspects to be faulty. We make the assumption that protocol objects on the same network node do not fail *independently*.

Distributed Agreement Service. The need of having distributed objects to agree on some common value is central to many problems in fault-tolerant distributed computing. So, robust consensus related protocols

are the cornerstone of most fault-tolerant systems. In BAST, protocol classes `CSSObject` and `DTMObject` both provides operations capable of solving the distributed agreement problem, even when some faulty nodes are involved. Class `CSSObject` enables to reach distributed agreement on any object as value, and each participant can start the consensus protocol (we say it is a symmetrical protocol).

Class `DTMObject` is at the heart of BAST's main distributed agreement pattern: the *Dynamic Terminating Multicast* (DTM) [19]. With class `DTMObject`, an initiator first multicast an object to all participants, each of which sends it reply to all others. Distributed agreement is then reached on a *collection* containing a subset of those replies. In Section 5, we show in details how a general atomic commitment and a total order multicast can be build by customising the DTM distributed pattern. Note that Bob is normally not supposed to use DTM directly, although nothing prevents him from doing so.

Atomic Group Service. When Bob wants to insure fault-tolerant total ordering of messages on a group[5] of protocol objects, he has the choice between two protocol classes, namely `ABCObject` and `TOMObject`. The difference can be expressed as follow: when member of two distinct groups, a `TOMObject` insures total ordering of messages sent to either groups, whereas a `ABCObject` only insures total ordering of messages that are sent within each group. In other words, if messages m_1 and m_2 are sent in two distinct yet overlapping groups, they might be seen in different order by two objects of class `ABCObject`, but this is not possible for two objects of class `TOMObject`. As we shall see in next section, class `ABCObject` is well-suited for insuring consistency of a replicated object. In Section 5, we present class `TOMObject` in details and we point out why it makes it easy to solve the *fault-tolerant distributed locking* problem.

Atomic Commit Service. Instances of class `ACMObject` are capable of solving various atomic commitment problems, depending on the semantics required by Bob. In order to support a variety of such semantics, an additional class hierarchy is provided, of which the root abstract class is `AbstractTransaction`. BAST already offers two kind of ready-to-use transactions classes, namely `NonBlockingTransaction` and `ReplicatedTransaction`.

3 Using BAST

In BAST, distributed protocols are said to be *generic*. In this context, genericity implies that the type of arguments passed to any protocol operation is not restricted; e.g., operation `rSend()` can be used to send any object across the network[6]. 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 organised into a single inheritance hierarchy: each protocol class implements only one protocol, but instances of some π`Object` class can execute any protocol inherited from π`Object`'s superclasses (Figure 2b). So, protocol objects are capable of running several executions of identical and/or distinct protocols concurrently.

[5] Groups are only a logical addressing capability here, i.e., no group membership is assumed necessary.

[6] More precisely, we should say any object that is *not* a protocol object; sending protocol objects across the network requires to update all their distributed references before subsequent uses. Migrating protocol object is yet another distributed problem, but we do not deal with this issue here.

3.1 Achieving Active Replication

To illustrate how protocol classes can help Bob in building fault-tolerant distributed software, let's consider an application which objects are running on a network of four workstations, and suppose he wants some critical distributed object S to be available even when some workstations crash. One way to achieve this consists in *actively replicating* object S on all four workstations; in that case, we say that S is a replicated object and it is manipulated as a group of replica objects $g_s = \{S_1, S_2, S_3, S_4\}$. Whenever a workstation crashes, users working on the remaining three can go on with their work, because their local replica S_i is still available. Now the *liveness* of S is ensured, Bob still has to make sure all four replicas keep a consistent state throughout their execution, i.e., he is now concerned with the *safety* of S. To ensure consistency, he can use total order multicast communications on group g_s: by making sure all update messages are received by the four replicas in the same order, he knows their states will remain identical. What Bob really needs here is a protocol class that solves the *atomic broadcast*[7] problem; informally, the atomic broadcast requires that all correct replicas deliver the same update messages in the same order [5]. BAST provides such a protocol class, namely `ABCObject`, which defines operation `abcast()` and callback `abDeliver()`. By subclassing `ABCObject` and by making replicas of S instances of that new class, Bob can actively replicate S very easily: callback operation `abDeliver()` of such subclass is implemented in order to update local replica S_i in a deterministic way.

Figure 3 depicts what happens after some user action on workstation 1 generated an update for replicated object S. Instead of directly applying the update to local replica S_1, the application builds an update message `m` and invokes operation `abcast()` on S_1, passing it `m`. The responsibility of propagating the update is now delegated to S_1. Operation `abcast()` starts an execution of the atomic broadcast protocol involving replicas S_1, S_2, S_3 and S_4. Eventually, the protocol triggers operation `abDeliver()` on each replica, including S_1. This operation has been implemented in `ABCObject`'s subclass and performs the actual update, using message `m`. Because class `ABCObject` solves the atomic broadcast problem, Bob has the guarantee that any concurrent message will be identically ordered with respect to `m` on all replicas. So, Bob's job merely comes down to subclass `ABCObject` and to implement callback operation `abDeliver()`. Note that operations `abcast()` and `abDeliver()` can also be used to atomically transfer the most recent state to a replica that is recovering after a crash [20].

[7] The term "*broad*-cast" is used here (instead of "*multi*-cast"), because target group g_s is implicit for all update messages. Talking about "multicast" would suggest that each message could be sent to a different set of distributed objects, which is not the case in this example.

4 In-Depth View of BAST

Now we have seen Bob's view of the BAST framework (mainly as a library of ready-to-use protocols), next two sections are going to present another of its key features: its extensibility. This point of view concerns Alice, who has adequate skills in building fault-tolerant distributed protocols.

Before presenting how new protocol classes are actually added to BAST (Section 5), we first discuss some aspects of the way we model protocol relationships. In particular, we introduce the centralised *Strategy* design pattern and we show why it is at the heart of protocol composition in BAST. The *Dynamic Terminating Multicast* distributed agreement pattern (DTM)

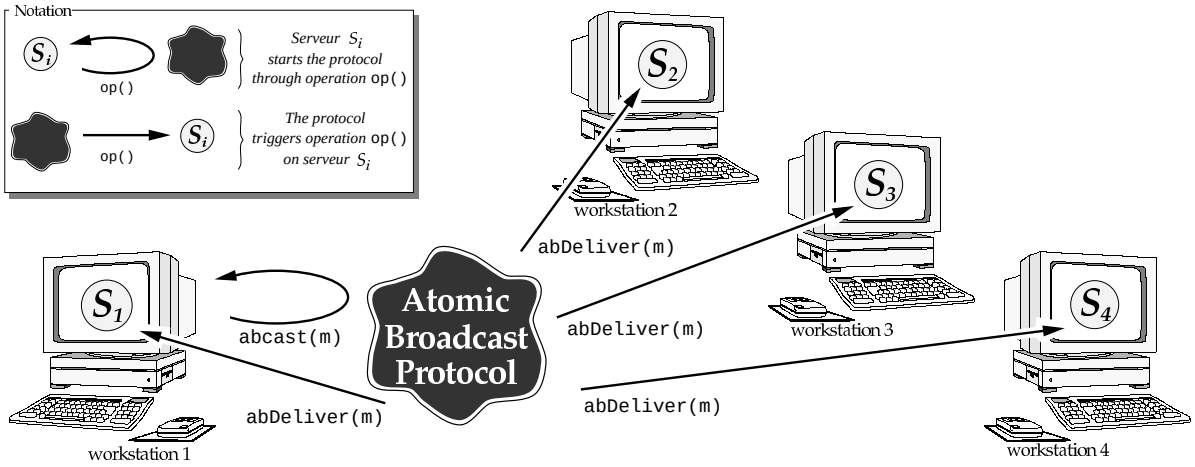


Figure 3: Active replication with class `ABCObject`

is extensively discussed in Section 5, together with two full examples of how it can be used; so, we do not present it here.

4.1 Protocol Dependencies

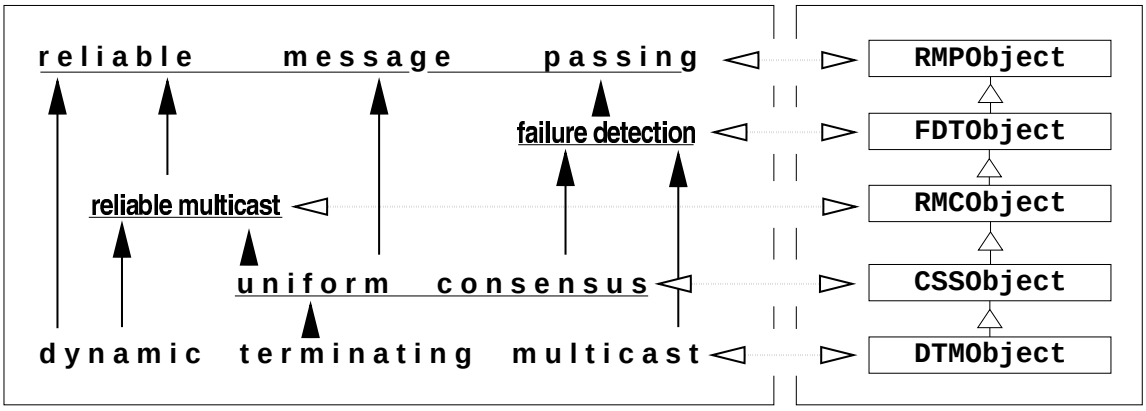
Fault-tolerant distributed protocols are challenging to build, because they involve many other underlying protocols. This can result in complex dependency relationships, as illustrated by Figure 4a. The DTM pattern implemented in BAST, for instance, is built on top of consensus, failure detections and reliable communications. Consensus is itself based on failure detections and on reliable communications, both point-to-point and multicast; in turn, BAST's reliable multicast relies on a reliable point-to-point communications. Figure 4b shows the corresponding protocol classes and their inheritance relationships, following the OMT notation.

In BAST, managing protocol dependencies is possible not only 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 5a presents an instance of class `ABCObject` during its execution. As we have seen, class `ABCObject` defines two entry operations to the atomic broadcast protocol: `abcast()` and `abDeliver()`. Besides those operations, protocol object `anABCObject` is also capable of concurrently executing any protocol inherited by its class, e.g., consensus and reliable multicast communications (Figure 2b). This is what makes the atomic broadcast easy to build.

Protocol Layers Interactions

In Figure 5a, `anABCObject` is simultaneously running five different protocols on behalf of the application layer, while performing low-level calls to the transport layer for this job. In Figure 5b, we focus on the protocol stack dedicated to some atomic broadcast execution. In BAST, protocol dependencies are modelled in a similar way to the *OSI* (Open Systems Interconnection) stack model [32]. In our first



(a) Protocol Dependencies (b) Protocol Class Hierarchy

Figure 4: Protocols and protocol classes in BAST

attempt to structure protocol interactions, we adopted a strictly vertically layered approach. However, we were forced to face the fact that it does not always make sense in terms of protocol dependencies, as suggested by Figure 5b. In the end, only the *encapsulation* benefit of the layered approach was retained, while its strictly vertical nature was discarded. Our conclusion here concurs the work done by Hüni and al. on their framework for network protocols [21]. Coming back to `anABCObject`, each time the application layer invokes one of its operations, a new protocol stack is created. So, each protocol execution is supported by an independent protocol stack, as suggested in Figure 5a. Furthermore, each layer of that stack is in charge of only one protocol supported by class `ABCObject`. Focusing on protocol object `anABCObject`, protocol composition means here to assemble various layers, each one running a protocol algorithm necessary to the execution of the atomic broadcast protocol. This is where protocol composition actually occurs, as we shall see now.

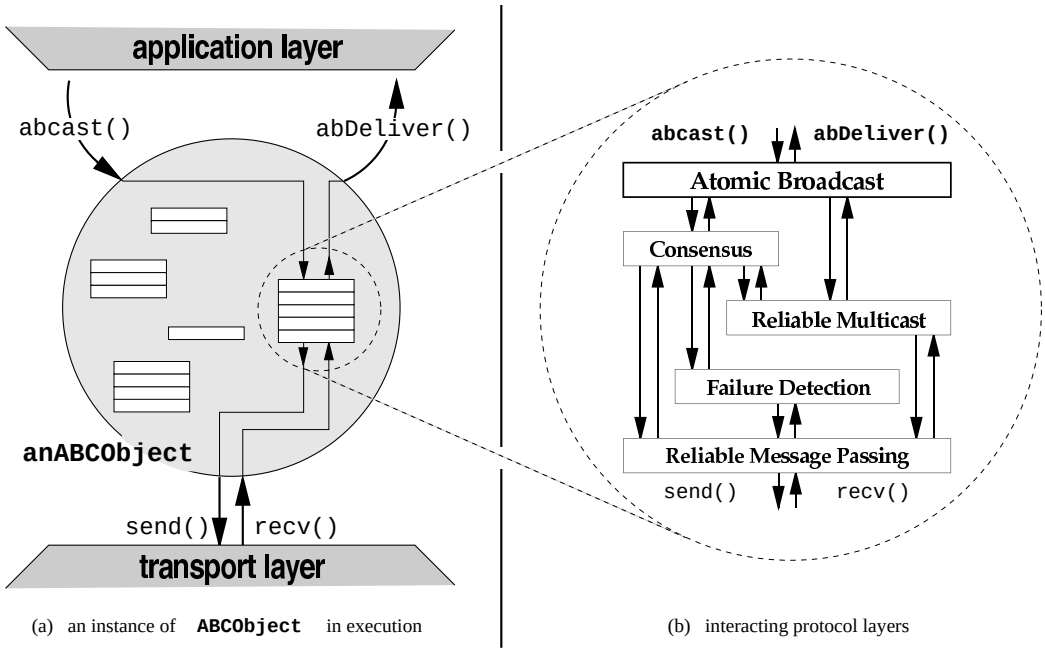
4.2 Algorithms as Objects

In order to separate protocol layers from their implementations, distributed algorithms are manipulated as separate objects. Consequently, each π`Object` protocol class is independent of the algorithm supporting protocol π, which makes protocol composition fully flexible. Distributed algorithm objects of class π`Algo` are privately used by the corresponding π`Object` protocol objects and hold the actual implementation of distributed algorithm π. An algorithm object always executes *within* a protocol object: as a result, Bob only deals with protocol objects and know nothing about algorithm objects, but Alice does.

Applying the Strategy Design Pattern

According to Gamma et al., the intent of the *Strategy* pattern is to "*define a family of algorithms, encapsulate each one, and make them interchangeable*" [8, page 315].

Making each π`Object` protocol class independent of the algorithm supporting protocol π is precisely what we provide for composing reliable distributed protocols in a flexible



(a) an instance of **ABCObject** in execution

(b) interacting protocol layers

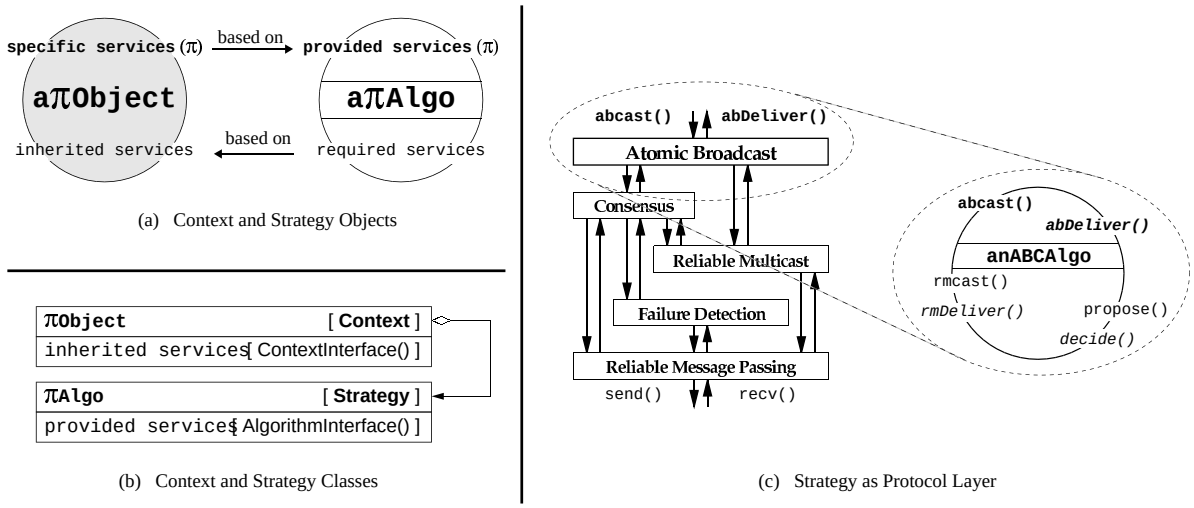
Figure 5: Protocol objects and the layered model

manner. In other words, the Strategy pattern is a key element in BAST, as far as flexible protocol composition is concerned. This design pattern is usually implemented by objectifying the algorithm [7], i.e., by encapsulating it into a so-called *strategy* object; the latter is then used by a so-called *context* object.

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 the *strategy*, and a protocol object, instance of some π`Object` class, is the *context*. A strategy and its context are strongly coupled and the application layer only manipulates instances of π`Object` classes.

Figure 6a 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, aπ`Object` uses services provided by strategy aπ`Algo` (right side of Figure 6a). 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 identify entry point operations to underlying protocols needed to solve problem π. Figure 6b presents the relationship between classes π`Object` and π`Algo`, using a class diagram based on the OMT notation.

Strategy/Context Interactions. The way aπ`Object` and aπ`Algo` cooperate can also be



(a) Context and Strategy Objects

(b) Context and Strategy Classes

(c) Strategy as Protocol Layer

Figure 6: Strategy design pattern in BAST

seen as some kind of *symbiosis*: two dissimilar objects participate to a mutually beneficial relationship. Instances of class π`Algo` each represent a distinct execution of protocol π's algorithm. So, another way to look at algorithm objects can be expressed as follows: at runtime, each algorithm object represents a layer in one of the protocol stacks currently in execution within aπ`Object` (Figure 6c).

Flexible Protocol Composition

The Strategy/Context separation offers full flexibility in protocol composition and makes it possible to change any protocol algorithm of any protocol class very easily. Protocol dependencies are expressed only at the specification level, through subclass relationships, whereas the implementations can vary. One could for example optimise the reliable multicast algorithm on which class `ABCObject` relies, while leaving it unchanged in other classes. Protocol algorithms could even be dynamically edited and/or chosen, according to criteria computed at runtime; this feature is analogous to the dynamic interpositioning of objects.

Another advantage is that Alice can strictly focus on protocol π and on the interaction between class π`Object` and class π`Algo`: she need not to know how other protocols are implemented. Because aπ`Object` must be able to execute all lower-level protocols on which protocol π is based, Alice is also forced to clearly *specify* what underlying protocols are required to implement corresponding protocol π. She can then choose which protocol class to subclass in order to implement π`Object`. A complete step-by-step methodology for extending BAST through protocol composition can be found in next section.

Concurrent Protocol Executions

Protocol algorithm classes define a completely separate inheritance hierarchy from the protocol objects' one (with `ProtoAlgo` as root). Each instance of some π`Algo` class 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 aπObject. At runtime, the latter maintains a table of all strategies that are currently in execution for it. Each message is tagged to enable aπObject to identify in which execution of what protocol that message is involved, and to dispatch it to the right strategy. As a consequence, concurrent protocol executions are managed in a straightforward manner.

Recursive Use of Strategies

As already pointed out, Alice can strictly focus on protocol π and on its associated πObject and πAlgo classes. In particular, all protocols needed to support protocol π are hidden in inherited operations. The latter might also be implemented applying the Strategy pattern, but this is transparently managed by superclasses of πObject. In that sense, BAST uses the Strategy pattern in a powerful *recursive* manner.

The recursive use of the Strategy pattern is illustrated in Figure 7. The latter schematically presents a possible implementation of protocol class CSSObject, which enables the solution of the consensus problem by providing operations propose() and decide(). In Figure 7, the gray oval is context class CSSObject, while inner white circles are various πAlgo strategy classes. 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 7 is managed by the corresponding context class in the protocol class hierarchy presented in Figure 2b.

Drawbacks and Limitations

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/or complex protocols are involved (including marshaling). 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[8] (without even considering failures or marshaling time). The gain in flexibility clearly overtakes the local overhead caused by the use of the Strategy pattern.

There is also a minor compatibility constraint among different protocol algorithms in order to make them interchangeable: some new algorithm class πAlgo$_n$ can replace default πAlgo *if and only if* πAlgo$_n$ requires a subset of the services featured by the hosting protocol class.

A more detailed discussion on the use of the Strategy design pattern in BAST can be found in [13].

5 Extending BAST

There are basically two methods for Alice to create protocol classes with BAST: (1) by customising a protocol class implementing a *distributed pattern*, such as DTM, or (2) by composing existing protocol classes to build a brand-new one. The first method is adequate when the newly built protocol can be seen as an instance of another more general protocol (the distributed pattern); in this case, we talk about *protocol customisation*. This feature is what makes BAST a framework in the sense Campbell et al. define it [4]. The second method

[8] On a 10 Mbits Ethernet connecting Sun SPARCstations 20.

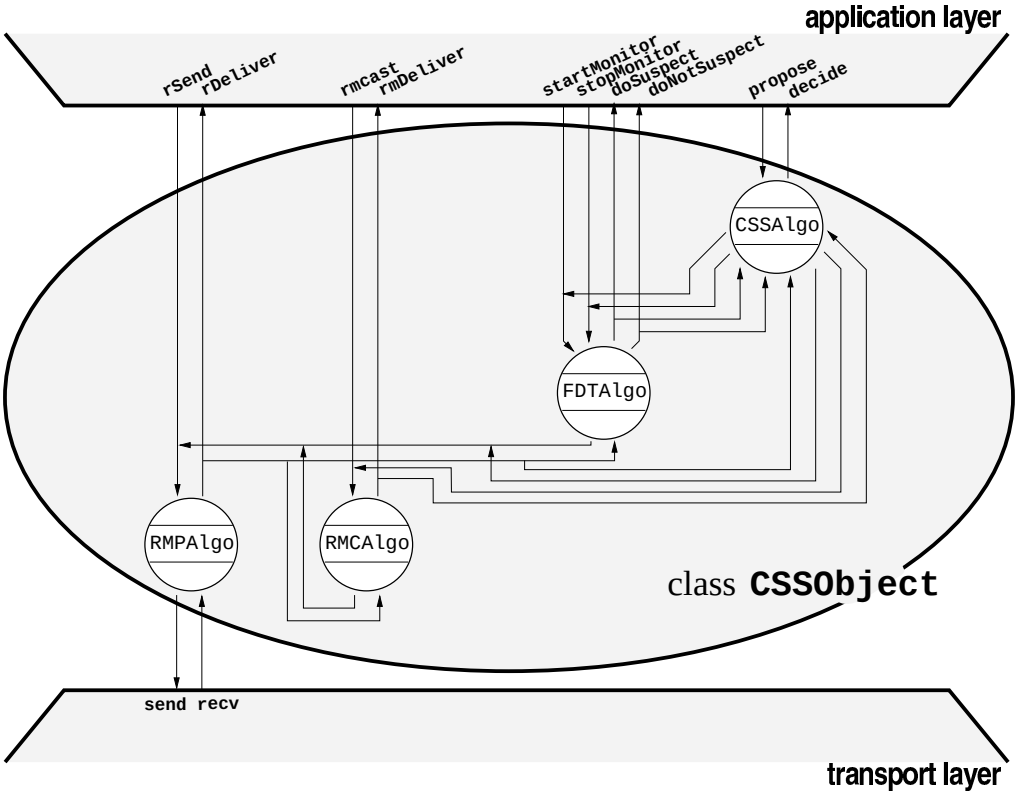


Figure 7: Recursive use of the Strategy pattern

is adequate when the new protocol cannot be built with the first method, but relies on protocols that are already available in BAST.

With BAST, it is also possible to *tune* any existing protocol class by changing one or more distributed algorithms it relies on. In that case, Alice does not really extend BAST, but this is an essential feature in adapting protocol classes to specific environments and contexts. Next three subsections discuss how to customise a distributed pattern, how to build a new protocol by composing existing ones, and how to tune a protocol by changing some underlying distributed algorithm.

5.1 Applying the DTM Pattern

We now illustrate distributed pattern customisation in BAST by showing how Alice can instantiate the *DTM agreement pattern* to produce either an non-blocking atomic commitment or a fault-tolerant total order multicast. Since both protocols require the solution of the fault-tolerant consensus, the DTM pattern is perfectly suited here. In this context, the verb "to instantiate" does not mean "to create an instance of some class", as in the object-oriented terminology, but to "apply the DTM distributed pattern".

Distributed Agreement Pattern

The DTM distributed pattern enables an *initiator* object to reliably multicast a message m to $Dst(m)$, a destination set of remote participant objects, and to reach agreement on $Reply(m)$, a set of replies $reply_k$ returned each $participant_k$ in response to m. This is why we say it is a *distributed agreement* pattern.

In order to customise the DTM distributed pattern, Alice has to subclass protocol class `DTMObject`. Instances of such subclasses can play the role of both the *initiator* and the *participant*. Figure 8 presents what objects and operations are involved while the protocol is executing: fat arrows picture operation invocations on objects, bullet-arrows ($\leftarrow\bullet$) represent objects resulting from invocations, and numbers in circles show in what order invocations occur. The *initiator* object is on node A, while $participant_i$ and $participant_j$ are on node B and node C respectively; different nodes imply different address spaces. Since the interaction of the protocol with each participant is exactly the same, arrows are numbered for $participant_i$ only.

Overview of the Protocol The DTM pattern's protocol starts by the invocation of `dtmcast()` of an initiator object, passing it a message m, a set of remote participants objects $Dst(m)$ and a *validity conditions* (explained below); this invocation results in a reliable multicast to the set of participants. When message m reaches some $participant_k$, the latter is invoked by the protocol through the `dtmReceive()` operation, taking m as argument. In turn, $participant_k$ computes and returns its $reply_k$. Eventually, each non-faulty participant is invoked through the `dtmInterpret()` operation with the $Reply(m)$ set, on which consensus has been reached, as argument. So, as long as `dtmInterpret()` implements a deterministic algorithm, all participants will take the same decision. Operations `dtmReceive()` and `dtmInterpret()` are invoked through callbacks by DTM.

Why is DTM a Pattern? As any protocol class in BAST, `DTMObject` is generic in the sense that message m sent by the initiator, the set of participants $Dst(m)$, the response $reply_k$ computed by each $participants_k$, and the interpretation of $Reply(m)$, the set of replies on which agreement is reached, are not defined *a priori*. More specifically, m and $reply_k$ can be any object, while the interpretation of $Reply(m)$ is performed by a callback operation.

However, DTM is said to be a distributed *pattern*[9] because it has something more: by customising its generic dimensions, one can produce new protocols, not only use it in some specific application. In other words, DTM is primarily intended to serve Alice, not Bob. A distributed pattern, such as DTM, can also be described as a *template protocol*, i.e., as some kind of *meta* or *abstract* distributed protocol, acting as template for application-level protocols aimed at Bob.

Validity Condition. One very important generic dimension to make DTM a distributed pattern is its *validity condition*, which allows to constrain $Reply(m)$: when that constraint cannot be satisfied, the protocol blocks. The reliability property of the DTM pattern lies in the fact that its protocol will not *necessarily* block if one or more participants fail[10] : it depends on the chosen *validity condition*. The $Reply(m)$ set received by all non-faulty participants might simply lack the replies of faulty

[9] In previous papers [9, 10, 11, 13], we refer to DTM as a *generic protocol*. Our point of view, as well as out terminology, has evolved since then.

[10] The protocol is based on (possibly unreliable) failure detectors to decide whether an object is faulty.

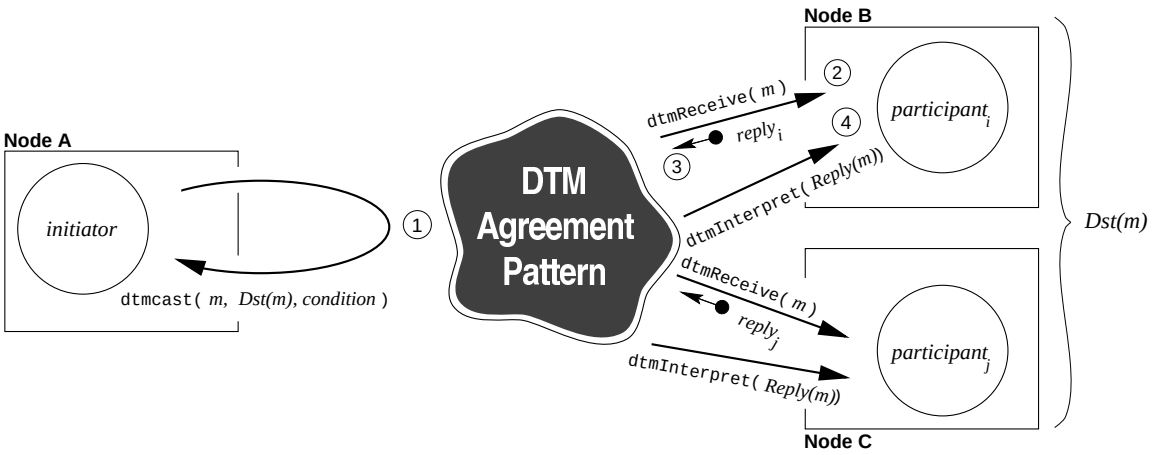


Figure 8: The DTM distributed pattern

participants. It is necessary to be able to express such a condition, since participants might fail and the $Reply(m)$ set might have a contents that does not permit to take any satisfactory "decision" (e.g., $Reply(m)$ could be trivially empty). An example of validity condition is the *majority condition*, that can be expressed as $|Reply(m)| > |Dst(m)|/2$, i.e., it requires a majority of non-faulty participants for the protocol not to block.

Operation `dtmcast()` of class `DTMObject` takes such a validity condition as third parameter. The latter can be any object, as long as it understands operation `test()`, which takes three sets of protocol object references as parameters. The validity condition is tested while the DTM pattern is collecting participants' replies, and is at the heart of reliability issues. When invoked by the protocol, operation `test()` receives three sets as arguments: $Dst(m)$, $Reply(m)$ and $Suspect(m)$, a subset of $Dst(m)$ which contains objects of $Dst(m)$ that are *suspected* to be faulty. Set $Suspect(m)$ is necessary for instances of the DTM pattern where failures have to be considered in the agreement process, i.e., when the condition on $Reply(m)$ is expressed in terms of failures.

Atomic Commitment with DTM

The *atomic commitment problem* requires that participants in a transaction agree on *commit* or *abort* at the end of the transaction. If participants can fail and Alice still want all correct participants to agree, the problem is known as the *non-blocking* atomic commitment ($NB\text{-}AC$) [1]. In that case, the agreement should be commit if and only if all participants vote *yes* and if no participant fails. It has been proved that this problem cannot be solved in asynchronous systems with unreliable failure detectors [18]. This lead to specify a weaker problem: the non-blocking *weak* atomic commitment ($NB\text{-}WAC$), which requires merely that no participant is *ever suspected*. Because the DTM pattern makes no assumption on the properties of the failure detector it uses, both the $NB\text{-}AC$ and the $NB\text{-}WAC$ problems can be seen as instances of DTM, depending on the failure detector considered.

Class `ACMObject`. To solve the atomic commitment problem using the DTM dis-

tributed pattern, Alice subclassesDTMObject into ACMObject. This class defines four new operations: requestCommitOf(), voteForCommitOf(), commit() and abort(); apart from the first one, all are triggered by callbacks. Class ACMObject also implements inherited operations dtmReceive() and dtmInterpret().

Operation requestCommitOf() initiates the protocol for a given transaction passed as argument (an instance of some AbstractTransaction's subclass). The protocol terminates when either commit() or abort() is triggered; those two operations are also passed the transaction which must be committed/aborted. Figure 9 gives an overview of the atomic commitment protocol based on DTM: on the far left, we picture a *transaction manager* that is willing to commit an atomic sequence of operations performed by the two *data managers* pictured on the far right. As in Figure 8, arrows represent invocations on objects, while numbers in circles show in what order invocations occur; we kept Arabic numerals for DTMObject-level operations and used Roman numerals for ABCObject-level ones.

We now sketch how objects interact while the atomic commitment protocol is executing. Figure 10 presents the implementation of the main operations involved. The pseudo-code used there is very simple: statements are separated by symbol ";", variables are untyped and declared as in "|| voteReq ||", the assignment symbol is "←", and the value returned by an operation is preceded by symbol "↑".

Initiator Side. When the transaction manager wants to commit an atomic sequence of operations that occurred during a given transaction, it invokes operation requestCommitOf() on its ACMObject, passing it the corresponding transaction object. Transaction objects are instances of AbstractTransaction's subclasses and hold all relevant information for starting atomic commitment protocols.

Operation requestCommitOf() first creates a VoteRequest message, and stores it in a local variable voteReq (Figure 10a). Class VoteRequest defines instance variable transaction, which holds the actual transaction object. Then, operation requestCommitOf() starts the atomic commitment protocol by invoking inherited operation dtmcast(). Generic set $Dst(m)$ and the generic *validity condition*, passed to dtmcast() as second and third arguments respectively, are transaction's instance variables.

Participant Side. When message voteReq reaches a participant ACMObject, callback operation dtmReceive() first extracts the transaction object (Figure 10b). It then passed the transaction to operation voteForCommitOf() defined by class ACMObject, which is expected to compute a vote concerning the commitment of the transaction (either *yes* or *no*); the vote is returned to DTM.

The DTM distributed pattern then collects the votes of all non-faulty participants and put them into voteSet, a set implementing generic set *Reply(m)*. During this collecting phase, the validity condition may be tested several times. As soon as voteSet satisfies operation test() (Figure 10c), the DTM pattern starts an agreement protocol by invoking inherited operation propose(), with voteSet as argument. Eventually, callback decide() is triggered on each (non-faulty) participant. DTMObject's implementation of operation decide() merely calls dtmInterpret() and passes it the value on which agreement has been reached, namely voteSet. Operation dtmInterpret() then computes the

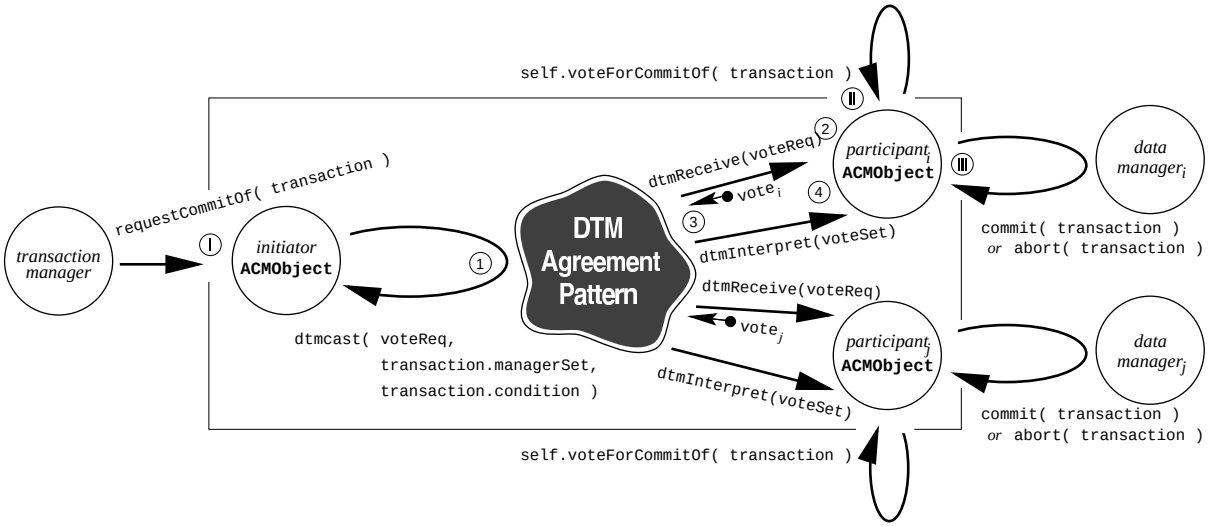


Figure 9: Atomic commitment with DTM - overview

final decision, which is *commit* only if all votes in `voteSet` are *yes* and if no participant `ACMObject` was suspected. In order to do that, `dtmInterpret()` first extracts the `transaction` object from any vote found in `voteSet` (Figure 10d). It then compares the size of `voteSet` with the size of `transaction.managerSet`: if both have the same size, it means that no participant was suspected by DTM. It then goes through each vote to see if the all are *yes*. Operation `dtmInterpret()` ends up by invoking either `commit()` or `abort()`, passing it the transaction; those operations are expected to act on the corresponding data manager consequently.

Validity Condition. The semantics supported by object `transaction` directly depends on its validity condition, held in variable `transaction.condition`. For example, instances of `ReplicatedTransaction` hold a majority condition, whereas instances of `NonBlockingTransaction` hold the condition implemented by operation `test()` of Figure 10c. This operation is expressed in terms of participant failures and implements the following predicate: $\forall participant_k \in Dst(m) : reply_k \notin Reply(m) \Rightarrow participant_k \in Suspect(m)$. Interestingly, the same condition is suitable for both *NB-AC* and *NB-WAC* problems, because it all depends on the actual failure detector the DTM pattern is using.

Total Order Multicast with DTM

The *reliable total order multicast* problem can be specified by two primitives, *TO-multicast(m, Dst(m))* and *TO-deliver(m)*, and by a set of conditions on those primitives. Those conditions express that consistency and liveness must be preserved despite object failures, and that if more than one object are in the intersection of several different $Dst(m)$ sets, they must all perform the corresponding *TO-deliver()* in the same order. Note that we are not talking of some broadcast primitive here: the *TO-multicast(m, Dst(m))* primitive requires the destination set *Dst(m)* to be

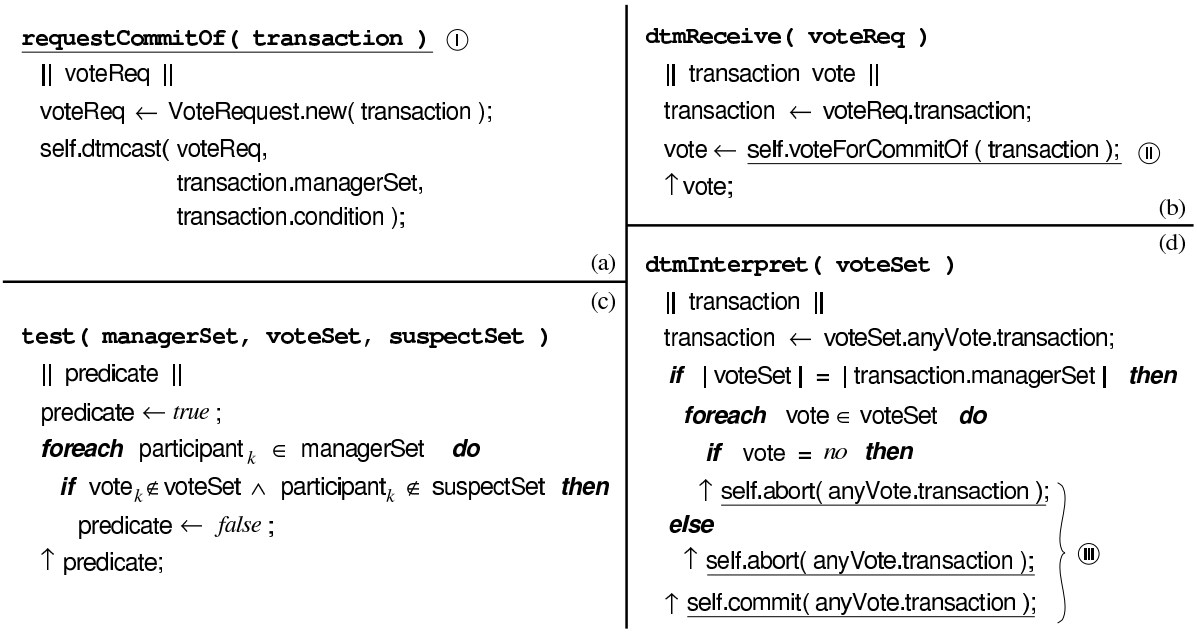


Figure 10: Atomic commitment with DTM - details

explicitly specified and that set can be different for each invocation[11]. This is why the order condition is expressed in terms of several $Dst(m)$ sets. A formal definition of the total order multicast problem can be found in [34]

To our knowledge, the algorithm presented here is the first *genuine multicast* protocol capable of insuring total order in a distributed system with unreliable failure detectors. We use the term "*genuine multicast*" to indicate that the algorithm is not trivially based on a broadcast protocol (which would not be scalable). This algorithm has been described and proved elsewhere [34], without using the DTM distributed pattern. Since it is quite a complex protocol, we first present it independently of DTM. Note that such a total order multicast protocol makes it straightforward to solve the distributed locking of several replicated servers, even when some replica fail.

[11]In a broadcast primitive, the destination set is implicit: it is the whole system (Section 3).

Overview of the Protocol. The basic idea of the algorithm presented here is to have each object in $Dst(m)$ to propose a time-stamp for message m, and to reach an agreement on the maximum of those time-stamps; the latter is then used as sequence number for message m, and messages are delivered according to their sequence numbers. Time-stamps are based on *Lamport's logical clocks* [23]. So, when object o receives message m, it sends its current logical clock value as proposed time-stamp to all other objects in $Dst(m)$. It then stores m in a queue of *pending messages*, i.e., all messages in that queue do not have their sequence number computed yet. When agreement is reached on m's sequence number, object o moves m from the queue of pending messages to a queue of *delivery messages*, i.e., all messages in that queue do have their associated sequence number but

have not been delivered yet. Finally, object o performs *TO-deliver(m)* for each message m in the delivery queue which sequence number is smaller than the proposed time-stamps of all messages in the pending queue.

Three additional conditions have to be fulfilled for the protocol to work: (C_1) *causal order* delivery must be ensured for all messages exchanged in the algorithm; (C_2) each logical object o in $Dst(m)$ has to be replicated and its replication rate must be such that there is always a majority of correct replicas of o in the system; (C_3) the sequence number has to be the maximum of the time-stamps that have been proposed by a *qualified majority* of replica objects in $Dst(m)$.

Condition C_2 leads $Dst(m)$ to contain groups of objects (replicas) rather that individual objects, each group gathering the replicas of one logical object. The notion of group is used here merely as a *naming facility*, i.e., no group membership protocol (as in Isis [3]) is necessary for the algorithm to be correct[12]. The qualified majority of $Dst(m)$ is a set of objects that contains a majority of replicas of *every group* in $Dst(m)$. So, condition C_3 can be expressed as the following predicate: $\forall g \in Dst(m) : |tsSet_g| > \frac{1}{2} \times |g|$, where $tsSet_g$ is the set of time-stamps proposed by replica objects in group g, when m's sequence number is computed. It is beyond the scope of this paper to explain why those additional conditions are necessary; details can be found in [34].

Class `TOMObject`. To implement the total order multicast protocol presented above, using the DTM distributed pattern, Alice subclasses `DTMObject` into class `TOMObject`. This class defines new operations `tomcast()` and `toDeliver()`, which implement the total order multicast primitives *TO-multicast()* and *TO-multicast()* respectively. Class `TOMObject` also implements inherited callbacks `dtmReceive()` and `dtmInterpret()`.

From clients' point of view, instances of class `TOMObject` represent some multicast service which allows them to initiate multicasts of totally ordered messages on *explicit* sets of replicated objects; this is precisely why the solution of distributed locking problem is straightforward with class `TOMObject`. On the server side, each replica's has an associated `TOMObject` which acts as intermediary and participates in the protocol for it, i.e., `TOMObject` instances are in charge of computing sequence numbers for messages and reordering them accordingly. Figure 11 presents an overview of the total order multicast based on the DTM pattern, following the same conventions as Figure 9. In Figure 11, $participant_{A_i}$ and $participant_{B_j}$ are two instances of class `TOMObject` managing replicas i and j of *server A* and *server B* respectively. When operation `toDeliver()` is triggered on a *participant* `TOMObject`, the message passed as argument is forwarded to the corresponding *server replica*, with the guaranty that total order is satisfied.

We now sketch how objects interact while the total order multicast protocol is executing. Figure 12 presents the implementation main operations involved; the pseudo-code is the same as in Figure 10.

Initiator Side. When some client object wants to issue a total order multicast to a set of replicated server objects, it first has to build a set of groups, each containing the replicas' distributed references (instances of `ProtobjectRef`) of a logical server. Such groups of replicas might for example be provided by some separate *naming service*. The client then invokes operation `tomcast()` on

[12] In this context, interpret "*object* $o \in Dst(m)$" in interpreted as "*object* $o \in$ *group* $g \wedge$ *group* $g \in Dst(m)$".

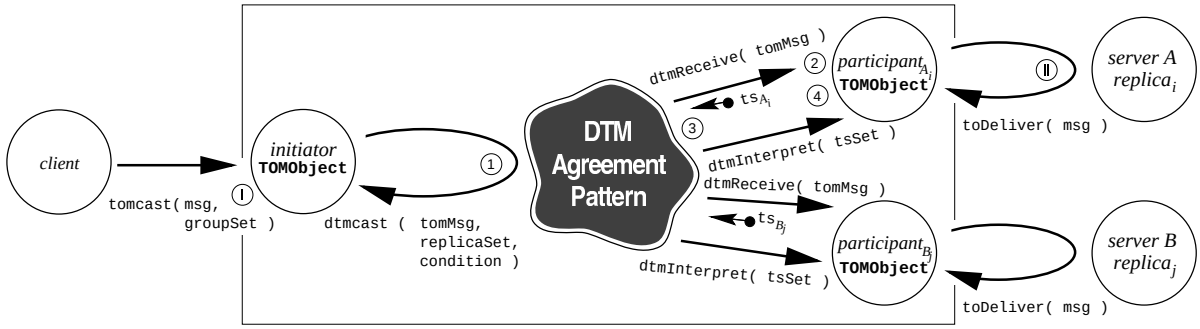


Figure 11: Total order multicast with DTM - overview

the initiator `TOMObject`, representing the multicast service. Two arguments are passed to operation `tomcast()`: `msg`, which can be any object, and `groupSet`, the set of groups it just built.

Operation `tomcast()` is implemented as follow (Figure 12a): (1) internal message `tomMsg`, instance of class `TOMMessage`, is created and its variable `toMsg.msg` is initialised with original message `msg`, (2) all groups of replicas are merged into `replicaSet`, a single set of protocol object references, (3) a validity `condition` is created as an instance of class `TOMCondition`, (4) finally, `dtmcast()` is invoked, with `tomMsg`, `replicaSet` and `condition` as arguments.

Participant Side. Class `TOMessage` also defines variable `toMsg.ts`, which is used by each participant `TOMObject` to hold the proposed time-stamp when `toMsg` is stored in the pending queue, and `toMsg`'s sequence number when it is stored in the delivery queue. The pending queue and the delivery queue are held in `TOMObject`'s instance variables `pendingQueue` and `deliveryQueue` respectively. So, when message `toMsg` reaches some participant `TOMObject`, operation `dtmReceive()` updates its Lamport's clock, sets `toMsg.ts` to the updated logical time and stores `toMsg` in the `pendingQueue` (Figure 12b). It then returns `toMsg.ts` to the DTM pattern. Variable `toMsg.ts` contains an instance of class `TimeStamp`, which implements generic $reply_k$ as Lamport's logical time.

The DTM distributed pattern then collects time-stamps and put them into `tsSet`, a set implementing generic set *Reply(m)*. During this collecting phase, the validity condition may be tested several times. As soon as `tsSet` satisfies operation `test()` (Figure 12c), the DTM pattern starts an agreement protocol by invoking inherited operation `propose()`, with `tsSet` as argument. Eventually, callback `decide()` is triggered on each (non-faulty) participant. `DTMObject`'s implementation of operation `decide()` merely calls `dtmInterpret()` and passes it the decision on which agreement has been reached, namely `tsSet`. Operation `dtmInterpret()` then computes `toMsg`'s sequence number, moves `toMsg` from the `pendingQueue` to the `deliveryQueue`, and performs `toDeliver()` for all original messages `toMsg.msg` that have been made *deliverable* by the newly computed sequence number (Figure 12d). Operation `dtmInterpret()` relies on private operations `getMsgRelatedTo()`

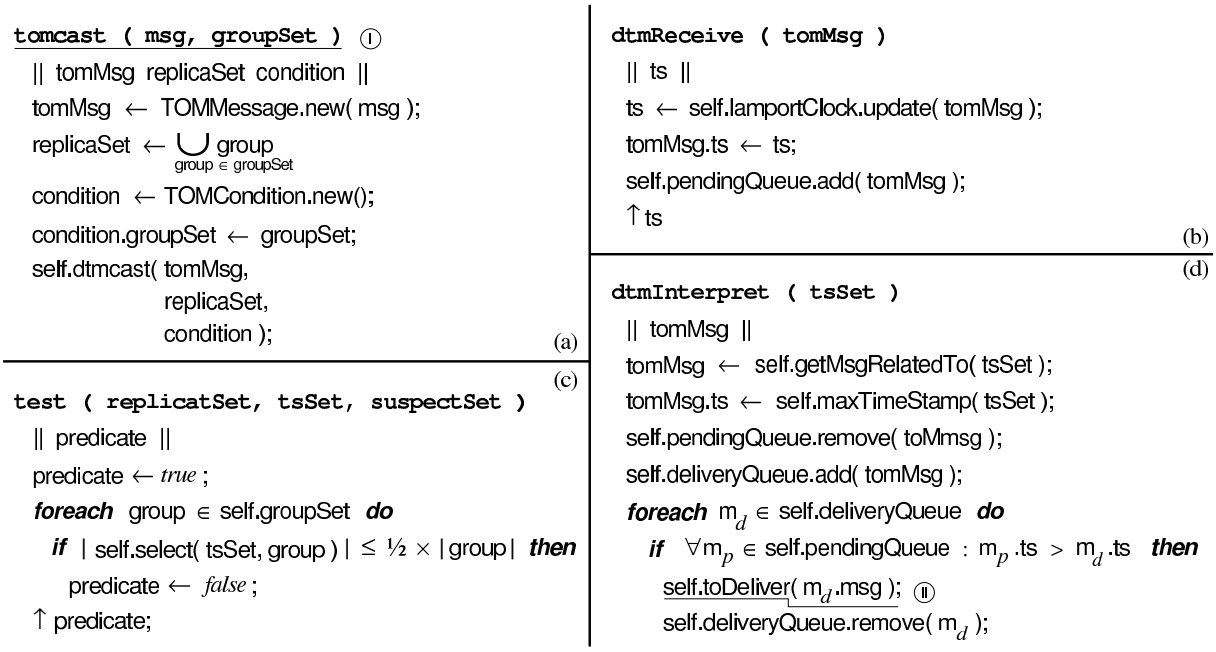


Figure 12: Total order multicast with DTM - details

and `maxTimeStamp()`, defined by class `TOMObject`. Those two operations allow to get the message associated with `tsSet`, and to compute the maximum time-stamp in `tsSet` respectively.

Validity Condition. Class `TOMCondition` implements the validity condition of the total order multicast protocol, i.e., its `test()` operation evaluates condition C_3 presented earlier, based of the notion of qualified majority. In that condition, *Dst(m)* contains groups of objects rather than individual objects. So, `TOMCondition`'s implementation of `test()` simply ignores the first argument passed to it by DTM[13] (Figure 12c). Instance variable `groupSet` is used instead, which holds the set of groups that was passed to operation `tomcast()` by the client. Private operation `select()`, defined by class `TOMCondition`, extracts from `tsSet` the proposed time-stamps of a particular `group` and puts them in a new set.

[13]The first argument of `test()` is a set containing references to protocol objects, not references to groups.

5.2 Protocol Composition

As already said, basing the BAST framework on the Strategy pattern has the advantage of making protocol composition very easy. We now present a five steps methodology which guides Alice in extending BAST; to illustrate this point, we present how she would build the `DTMObject` protocol class. Figure 13 summarises the methodology.

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

2. Choose an algorithm implementing DTM and determine what services it requires, by decomposing it in a way that allows to reuse as many existing protocols as possible; those services are: consensus, failure detections, as well as reliable point-to-point and reliable multicast communications (see [19] 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`.

5.3 Protocol Tuning

Class `CSSObject` is at the heart of the DTM distributed pattern, and hence at the heart of all agreement protocols that are built out of class `DTMObject`. This depicts the fact that in our design, consensus is considered a fundamental building block in the design of agreement protocols [19].

The actual consensus algorithm, used by class `CSSObject`, is encapsulated by class `CSSAlgo` (following the Strategy Design Pattern). The consensus algorithm we have considered in our current implementation is that of Chandra and Toueg [5]. This algorithm tolerates unreliable failure detection (which is typical in practical distributed systems), and non-blocking, in the sense that any correct process eventually decides despite the crash of other processes. In runs where there is no process crash or failure suspicion (the most frequent runs in practice), the algorithm requires 3 communication steps and $3n$ messages to reach consensus, where n is the number of participants in the protocol. Alice could actually be willing to trade efficiency with fault tolerance, and consider a blocking version of consensus, which requires only 1 communication steps and n messages to reach a decision. There are indeed applications, or system configurations, for which the probability of having a crash *during the algorithm execution* is considered small enough to be neglected. Such optimisation implicitly impacts on the performance of classes `DTMObject`, and hence on classes `ABCObject`, `TOMObject` and `ACMObject`. Interestingly, the resulting algorithm for class `ACMObject` comes down to be the famous *Two Phase Commit* algorithm, which is indeed blocking (non-fault-tolerant) [1], and the resulting algorithm for class `ABCObject` comes down to be the old Isis total order algorithm [3]. Note that this is not surprising, since our aim with BAST was not to develop new agreement algorithms, but rather to design a framework for expressing these algorithms in a modular way.

6 Implementation Issues

Our first prototype of the BAST framework was implemented using VisualWorks 2.x, the commercial Smalltalk [15] platform by ParcPlace-Digitalk Inc. The development took place on a network of Sun SPARCstations 20 and Ultra 1, running Solaris 2.x operating system. It was recently ported to Java, using Sun's Java Development Kit (JDK) on the same hardware/OS environment.

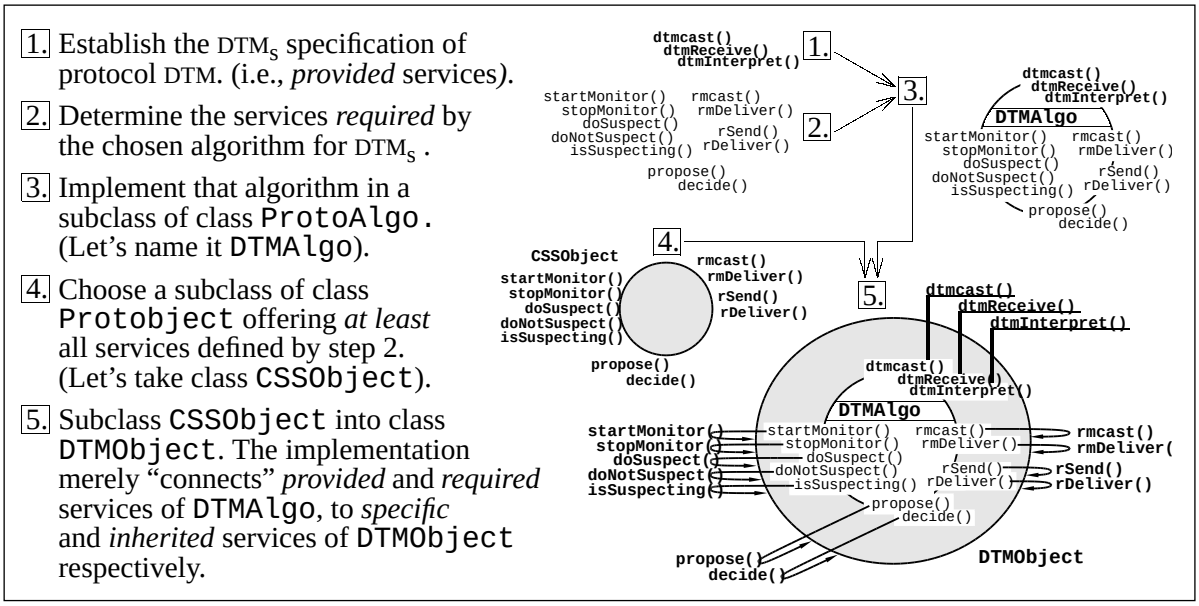


Figure 13: Extending BAST with protocol class DTMObject

Because Smalltalk and Java are "semantic twins", BAST's design is virtually the same in both languages. The only differences have to do with dynamic type checking in Smalltalk vs. static type checking in Java, and with their respective standard libraries. None of those differences are relevant here. In the following, we discuss the overall architecture of BAST independently of the language, together with some key issues in design and implementation. Performance results of the Smalltalk implementation are also presented; preliminary tests tend to suggest that performance results of the Java version would be very similar.

6.1 Implementation Overview

As already discussed in Section 2, protocol classes are what Bob is given to build reliable distributed applications. Alice has to deal with an additional kind of classes: protocol algorithm classes, which are all subclasses of abstract class ProtoAlgo.

We now give an overview of how BAST's basic services have been implemented in Smalltalk, except for the *Atomic Group* and the *Atomic Commit* services, which have already been extensively discussed in Sections 3, 4 and 5.

Communication Classes

All point-to-point communications, provided by classes UMPObject, BMPObject, RMPObject, FMPObject, are performed by *private* class TransportLayer, which represents the transport layer on each network node. In our implementation, a node is a *virtual machine* (VM, either Smalltalk or Java). Communications between VMs are implemented by class TransportLayer through a low-level layer written in C: *the Message Passing Layer*[14] (*MPL*, not pictured in Figure 2a).

[14] VisualWorks and JDK both allow to *dynamically link* C functions with Smalltalk, respectively Java, code.

MPL is based on UDP sockets and extends them with the delivery guarantees mentioned above.

All algorithms for point-to-point communications are implemented within MPL: class `TransportLayer` is only responsible for marshaling/unmarshaling messages and for dispatching them. As a consequence, none of the point-to-point communication objects uses private algorithm objects. However, MPL offers no support for reliable *multicast* communications. Class `RMCObject` is then based on algorithm class `RMCAlgo`, which implements a straightforward reliable multicast algorithm based on systematic retransmissions; details about this algorithm can be found in [5, 13].

Failure Detection Classes

As already said, we assume that `FDTObject` instances on the same network node do not fail *independently*, which in our implementation means that we only consider failures of VMs. The monitoring of remote VMs is performed for class `TransportLayer` by MPL. Class `TransportLayer` merely dispatches suspicions and non-suspicions to local protocol objects. MPL's failure detection scheme is implemented using timeouts and is assumed to behave as a failure detector of type $\diamond S$[15]. As for protocol classes supporting point-to-point communications, class `FDTObject` does not rely on an algorithm class: the failure detection protocol is wired in the underlying transport layer.

Distributed Agreement Classes

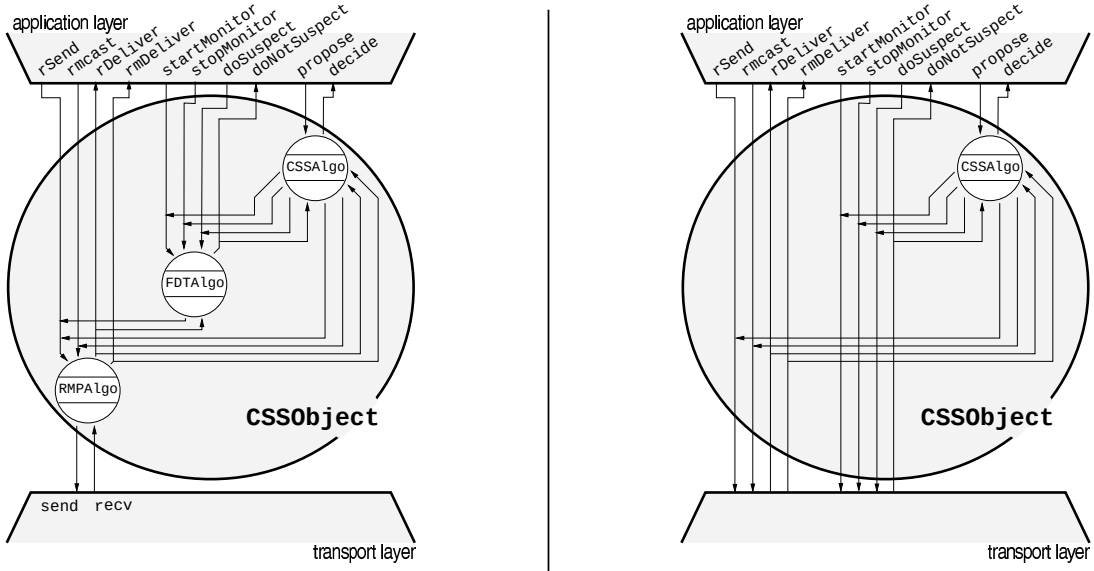
Protocol class `CSSObject` relies on algorithm class `CSSAlgo`, which implements the $\diamond S$-based consensus by Chandra and Toueg [5], while `DTMObject` is based on class `DTMAlgo`, implementing the DTM algorithm of Guerraoui and Schiper [19]. Both algorithms require protocol objects to perform reliable point-to-point and multicast communications, so `CSSObject` and `DTMObject` are subclass of `RMPObject` and `RMCObject`. Because objects participating in those protocols also have to monitor each other, classes `CSSObject` and `DTMObject` are subclasses of `FDTObject` as well. Finally, class `DTMObject` is a subclass of `CSSObject`, which factories the actual agreement protocol.

6.2 Transport Layer Dependency

Depending of the semantics of the underlying transport layer, protocol classes can be more or less complex. Figure 14 illustrates this point for class `CSSObject`: since transport layer in Figure 14a is *unreliable*, reliable communications have to be insured at the protocol class level, involving several different π`Algo` classes. In Figure 14b, on the contrary, transport layer supports reliable point-to-point communication, failure detection, as well as reliable multicast, so no algorithm object is needed for those protocols[16]. Having reliable communication algorithms wired in the transport layer reduces the complexity of class `CSSObject` and possibly increases performance. Furthermore, as long as the reliable transport layer still offers unreliable communications, no flexibility is lost: it is always possible for Alice to bypass wired-in reliable communications and re-implement them in adequate algorithm classes. This is also true for failure detection.

[15] Such failure detectors are said to be *unreliable* because they can make false suspicions; they have been proven to be sufficient to solve consensus in asynchronous systems [5].

[16] Our actual transport layer provides reliable point-to-point communication, failure detection but no reliable multicast. It is a tradeoff between Figures 14a and b.



(a) class **CSSObject** based on a unreliable transport layer

(b) class **CSSObject** based on a reliable transport layer

Figure 14: Dependence on the transport layer

6.3 Design Alternatives

We now discuss some design alternatives that we considered for implementing our first prototype of the BAST framework. Some of those alternatives have been part of an even earlier implementation but were later left out, while others are inspired by other existing frameworks.

Inheritance Alone

As presented in Section 3, inheritance is an appropriate mechanism to build fault-tolerant distributed applications from ready-to-use components: by subclassing appropriate protocol classes and by implementing their callback operations according to the desired semantics, Bob has the ability to tailor protocols to his needs. However, we pretend that inheritance alone is not sufficient as far as protocol composition goes, because it does not offer Alice enough flexibility. For example, inheritance does not allow her to easily implement a new algorithm for some existing protocol and to use it in whatever protocol class she wants.

We illustrate this situation in Figure 15: given the pictured class hierarchy, suppose Alice wants to change the reliable multicast algorithm on which both the causal and the atomic broadcast protocols are based (classes `CBCObject` and `ABCObject` respectively), while leaving it unchanged in all other classes (`CSSObject` and `DTMObject` in our case). With inheritance alone as code reuse mechanism, she has to implement the new reliable multicast algorithm in both `CBCObject` and `ABCObject` classes; this not what Alice expects from BAST as framework promoting flexible protocol composition and optimal code reuse. Furthermore, inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*. It is those limitations that lead us to *objectify* distributed al-

gorithms by recursively applying the Strategy design pattern, as presented in Section 5.

Mixins Classes

Assembling the various protocol layers through multiple inheritance is very appealing[17]: each π`0bject` class would implement only protocol π, while accessing all required underlying protocols through unimplemented operations. The latter would then be provided by other protocol classes through multiple inheritance. With such a design, protocol classes are all *abstract* and we usually talk about *mixin classes* or simply *mixins*.

There are several drawbacks with this approach, however. First, protocol classes are not more ready-to-use components: before being able to actually create a protocol object, one first has to build a new class deriving from all the necessary protocol classes. Bob no more can ignore protocol dependencies. Then, because protocol layers are assembled through subclassing, it is very difficult to compose protocols at runtime. While fairly complex yet possible in Smalltalk, such run-time protocol composition is impossible in Java (classes can only be created at compile-time in this language). Finally, concurrent protocol executions within the same protocol object have still to be managed, whereas this problem is handled naturally when each protocol execution is materialised as a separate algorithm object.

6.4 Secondary Design Goal

Apart from its modularity and extensibility, BAST was designed with the secondary goal of making complex fault tolerant distributed algorithms look as close as possible to their original form; such a feature is very interesting as far as teaching fault-tolerant distributed systems is concerned. Figure 16 illustrates this point: on the left, we placed an excerpt of the consensus algorithm proposed by Chandra and Toueg in [5], while on the right, we show how we implemented it in class `CSSAlgo`.

In Chandra/Toueg's paper [5], consensus is defined on some set Ω of distributed objects as follows: each object in Ω has the chance to propose a value and all correct objects eventually agree on one of the proposed values (the decision) [6]. Protocol class `CSSObject` defines operations `propose()` and `decide()`, which begin and terminate the protocol respectively [5]. Chandra/Toueg's algorithm is based on a rotating coordinator and assumes a failure detector of class $\diamond S$. In Figure 16a, the expression $n - f$ has to be greater or equal to $\left\lceil \frac{n+1}{2} \right\rceil$ for the algorithm to terminate, where n is the number of objects participating in the consensus ($n = |\Omega|$), and f is the number of faulty ones. Thus, the algorithm requires a majority (**maj** in BAST's code, Figure 16b) of correct protocol objects to be able to reach an agreement. Details about this algorithm, as well as a full discussion on the concept of failure detector can be found in [5]. As we shall see, being able to express distributed algorithms in such a "natural way" has an impact on performance.

6.5 Performance

As mentioned earlier, BAST was primarily built as an *open* environment for teaching and prototyping reliable distributed algorithms. In particular, all optimisations that would compromise its extensibility and genericity were left out, e.g., allowing *any* object to be sent across the network has a cost. As a consequence, BAST's first implementation is slow. However, now the first design and implementation of BAST is fully operational, we are currently achieving in-depth performance analysis

[17] Neither Smalltalk nor Java offer multiple inheritance. Smalltalk reflective facilities have been proven to make it possible to add multiple inheritance [22], while no such study has yet been published for Java.

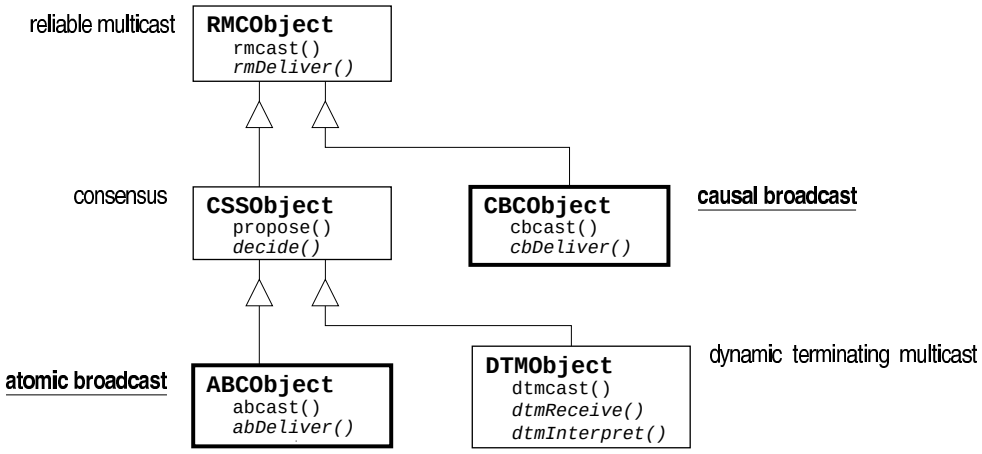


Figure 15: Problems with inheritance alone

in order to seek ways of optimising it.

In the rest of this section, we present some of our first performance results and conclusions, as far as potential optimisations go. Preliminary tests indicate that BAST on Smalltalk and Java offer roughly the same performance[18]; those tests also tend to prove that both versions could be optimised in a very similar way. In the following, we only discuss performance of the Smalltalk version.

Test Scenario

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

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

In this scenario, we consider a logical ring composed of three protocol objects located on three different network nodes, with each protocol object o_i knowing its preceding object $pred(o_i)$. Each object o_i sends its messages to all protocol objects in the ring (including itself), using some one-to-many communications operations (depending on the protocol being tested). Messages are said to be "empty", i.e., they hold a reference to the `nil` object, and they are sent in bursts of five, i.e., when protocol object o_i sends messages to all, it does so five times in a row. To implement fair accesses to the transport layer, each o_i waits until

[18] Compared to BAST built on JDK 1.0, the VisualWorks 2.x version was about 1.5 faster. With JDK 1.1, performance are now the same.

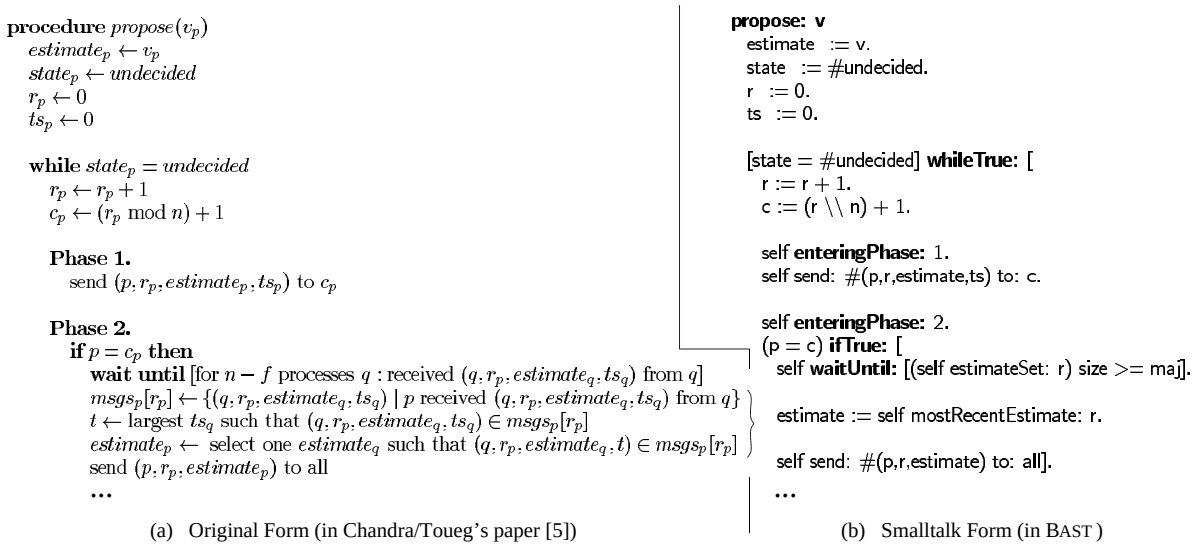


Figure 16: Making consensus look like consensus - excerpt

it x`Delivers` five messages from $pred(o_i)$ before sending another burst of five messages to all, with $x \in$ {`r, rm, abc`}. At each network node, time is started when o_i x`Delivers` its first message and stopped when it did so for 150 messages. By dividing 150 messages by the time o_i took to deliver them all, we get the throughput of each protocol for our scenario.

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

Performance Results

Figure 17 presents performance results for the scenario described above. The first column indicates which protocol is being tested, while the second column gives its throughput; performance worsen quickly when protocol complexity increases. Column 3 shows how many low-level messages are going through the transport layer (MPL) in order to deliver 150 high-level messages; column 4 shows how the average size of MPL-messages evolves. Although high-level messages merely contain the `nil` object, lower-level messages contain additional information related to the various underlying protocols; this information gets bigger as protocols become more complex.

Not surprisingly, 300 MPL-messages are processed by `RMPObject` objects performing reliable point-to-point communications: 150 are sent and 150 are received. Reliable multicast objects of class `RMCObject` process roughly 4 times more MPL-messages to do their job: each MPL-message received for the first time triggers three more reliable point-to-point communications (because of the systematic retransmission scheme described earlier). For `ABCObject` objects performing atomic broad-

protocol class	throughput [msgs/sec]	number of mpl-msgs	mpl-msg size [bytes]	marshaling [time percentage]	algorithms created
`RMPObject`	14	300	700	61	*one*
`RMCObject`	4	1173	1334	92	150
`ABCObject`	1	3406	2045	80	344

Figure 17: Performance when x`Deliver`*ing* 150 messages, $x \in$ {`r, rm, abc`}

casts, the number of MPL-messages is no more trivial to interpret: it includes many messages involved in underlying reliable multicast and consensus executions.

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

Possible Optimisations

As already suggested, optimisation should start at the marshaling level, in order to make it work faster and produce smaller messages[19]. This result is not surprising: current BAST's marshaling is based on VisualWorks *Binary Object Storage Service* (BOSS), which aims at making it easy to save any object onto stable storage[20]. Although BOSS semantics suits our need for marshaling, it was designed with very different performance goals in mind.

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

Performance of the MPL C Library

We also compared the performance of class `RMPObject`, which directly relies on MPL, with the latter used alone in a C program implementing the scenario described above (same network configuration, message size, burst size, etc.). With such a C pro-

[19] Producing a 700 bytes MPL-message for each "empty" high-level message sent through operation `rSend()` is far too much, even considering that such an MPL-message also contains the identity of the sender object.

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

gram, we get a throughput of 306 [msgs/sec] : MPL alone is then about 22 times faster than class `RMPObject`. This can be partly explained by the bad performance of BOSS-based marshaling, but more tests will be necessary to refine our analysis and to further optimise BAST.

7 Related Work

Object concepts are emerging as a major trend in distributed systems, and current research in object-based distributed environment follow several directions. Those research directions can be grouped into three main streams: (1) the extension of object-based languages, (2) the design of reflexive architectures, and (3) the definition of basic abstractions. The three research streams presented above are not competitive, but can be viewed as complementary ways to take benefits of object concepts in the context of distributed systems. For example, defining adequate abstractions is fundamental in order to take advantage of a reflexive architecture.

This paper has described a research work which belongs to the third stream: we are concerned in BAST with the design of distributed systems based on protocol classes as basic structuring abstractions. In the following, we compare our work with some previous approaches to build libraries and/or frameworks of distributed protocols. We mainly distinguish distributed transactions, group communications and network protocols.

7.1 Distributed Transactions

In [28, 30, 17], a transactional system is designed as a class library. Such a design leads to a better modularity, and enables to change the underlying transactional protocols with minimal effects on the rest of the system. Transactional components are first class citizens that can be customised according to the application's semantics.

Our work can be viewed as complementary to the research mentioned above, which does not discuss the way transactional protocols, such as distributed locking and distributed atomic commitment, are themselves designed and implemented on top of lower-level reusable components. Our approach precisely aims at providing a generic way to design and implement protocols, particularly *distributed agreement protocols* such as the atomic commitment, through adequate design patterns. The BAST framework does not address how to transparently attach transactional features to application objects.

7.2 Group Communications

Several authors have discussed the need for designing and implementing libraries of group communication protocols. The STREAMS framework [31], a pioneer in the domain, and the *x*-Kernel [29], contain rich libraries of communication protocols, but they do not deal with reliability and agreement issues. More recently, both in the context of the HORUS [35] and the CONSUL [25] projects, libraries or reliable distributed protocols were provided. The proposed approaches consider however the *group abstraction* as the basic abstraction for reliable programming, and hence limit the scope of both environments. Transaction-oriented applications are very difficult to support on top of such group-oriented systems, because protocols such as distributed locking and atomic commitment can hardly be designed on top of the *group abstraction* they provide; this is mainly due to the complexity of their underlying *virtual synchrony model* [2], which features full group membership management. In BAST, such a group membership is viewed as a specific agreement protocol, composed of lower-level protocols.

With BAST, we have tried to model *any kind* of interaction between distributed objects, not only group communications. This is essential in order to deal with failures in an extensible way, because reliable protocols tend to be much more complex than normal communications. By making protocol objects BAST's basic distributed entities, we can build both the group model and the transaction model [9]. Furthermore, the Strategy pattern provides a powerful scheme for creating new protocols through composition. Our BAST framework has a wider applicability as it is based on more basic abstractions such as reliable communication and failure detection. As we have shown, BAST goes a step further towards flexible protocol composition, by enabling different forms of protocol reuse, and providing the possibility to associate different implementations to the same distributed protocol.

7.3 Network protocols

CONDUITS+ offers basic elements that helps programmers build protocol layers. The use of design patterns is motivated by the fact that traditional layered architectures do not allow code reuse across layers, which is precisely what CONDUITS+ aims at. Protocols can then be composed with CONDUIT+, at lower-level than BAST, 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 the manipulation of protocol layers as objects, but only the manipulation of *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, strategies represent protocol layers, while conduits and information chunks 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[21]. With BAST, we are considering completely getting rid of inheritance but this issue has to be carefully studied, because it would have important consequences on the way BAST can be used by Bob, our emblematic application programmer who has limited skills in fault-tolerant protocols.

Microprotocols and the *x*-Kernel

The work done by O'Malley and Peterson [27] is the closest to BAST that we could find. They extended the *x*-Kernel with the notion of microprotocol graph, and they described a methodology for organising network software into a complex graph, where each microprotocol encapsulates a single function. In contrast, conventional ISO and TCP/IP protocol stacks have much simpler protocol graphs, with each layer encapsulating several related protocol functions. They argue that such a fine-grain decomposition allows for better tailoring of communication protocols to application needs; our conclusion concurs with theirs perfectly on that point. In their paper, O'Malley and Peterson mainly apply their approach to RPC communications (with only one very short discussion of what they call a *fault-tolerant multicast*). Compared to BAST, their approach is very close to what we have done and is based on the same basic assumption: composing (micro-)protocols is essential when it comes to customising complex distributed applications (and fault-tolerance implies such complexity). In their terminology, what we call

[21] 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.

problem π is referred to as *metaprotocol* π.

There are also some important differences, however. They do not provide ready-to-use protocol classes to programmers who are not skilled at understanding and/or building complex protocol graphs, whereas this is one of the main goals of BAST [12]. Moreover, their approach does not rely on design patterns. Similarly to CONDUIT+, they go one step further in their decomposition of protocol algorithms, by defining the notion of *virtual protocols*. The latter "*are not truly protocols in the traditional sense*" [27, page 131] : virtual protocols are actually used to remove IF-statements and to place them in the microprotocol graph instead. All those differences can be best understood by looking at the background domains of the BAST framework and the *x*-Kernel respectively. The latter aims at helping system programmers to customise any communication protocol usually found in modern operating systems, while the former aims at providing ready-to-use protocol classes, in order to help any programmer to build fault-tolerant applications, and at allowing skilled programmers to build news fault-tolerant protocols easily.

8 Concluding Remarks

We have presented BAST, an object-oriented framework for fault-tolerant distributed computing, based on centralised and distributed design patterns. BAST provides various ready-to-use protocols, e.g., the atomic broadcast and the atomic commitment, supporting the development of reliable distributed applications. It also offers the *Strategy* and *DTM* design patterns that are powerful tools to extend its protocol class library.

We described the way BAST was designed, and we pointed out issues in reusing and composing distributed protocols. In particular, we described in details how BAST's design patterns can be applied in order to create new protocols from existing ones. Further protocols, such as group membership or view synchronous communication, can then be easily developed, by using lower-level protocols provided by BAST.

BAST was first written in Smalltalk and then ported to Java. It has been used both for teaching reliable distributed algorithms and systems, and for prototyping new fault-tolerant distributed protocols. In order to make BAST useful for practical application development, we are currently working on performance testing and code optimisation. More information about BAST can be found on the web, at *http://lsewww.epfl.ch/bast*. Our first implementation is public-free and is also available there.

References

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

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

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

[4] R. Campbell, N. Islam, D. Ralia, and P. Madany. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993. Special Issue on Concurrent Object-Oriented Programming.

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

[6] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey).

Technical report, Department of Computer Science, Yale University, June 1983.

[7] 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.

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

[9] 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*, Linz (Autriche), July 1996. Springer Verlag.

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

[11] B. Garbinato, P. Felber, and R. Guerraoui. Right abstractions on adequate frameworks for building adaptable distributed applications. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 24–28. dpunkt-Verlag, 1997.

[12] B. Garbinato and R. Guerraoui. Flexible protocol composition in BAST. Technical report, Operating System Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology, March 1997. *Submitted to publication.*

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

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

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

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

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

[18] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In J.-M. Hélary and M. Raynal, editors, *Distributed Algorithms - 9th International Workshop on Distributed Algorithms (WDAG'95)*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100. Springer Verlag, September 1995.

[19] 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.

[20] R. Guerraoui and A. Schiper. Software based replication for fault-tolerance. *IEEE Computer*, pages 68–74, April 1997.

[21] 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.

[22] 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.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[24] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Koffmann, 1994.

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

[26] S. Mullender, editor. *Distributed Systems*. ACM Press, 1989.

[27] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

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

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

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

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

[32] M.T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, 1990.

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

[34] A. Schiper and R. Guerraoui. Faul-tolerant total order "multicast" with an unreliable failure detector. Technical report, Operating System Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology, November 1995.

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