# Protocol Classes for Designing Reliable Distributed Environments

Benoît Garbinato Pascal Felber Rachid Guerraoui

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

**Abstract.** In this paper, we present BAST, an extensible library of *protocol classes.* The latter is aimed at helping *system* programmers to build distributed programming environments. Protocol classes constitute the basic structuring components for higher-level programming models, such as the transactional model, and add flexibility to distributed environments. We focus on classes that implement a generic agreement protocol named *DTM* (Dynamic-Terminating-Multicast). To the programmer, the *DTM generic protocol* appears as a set of classes that can be specialized to solve agreement problems in distributed systems. In particular, we show how those classes can be derived to build *atomic commitment* and *reliable total order* protocols. An overview of the Smalltalk design and implementation of the BAST library is also presented.

## 1 Introduction

This paper describes BAST, an extensible class library of distributed protocols. BAST is aimed at assisting *system* programmers in building distributed programming abstractions for *application* programmers. It is more specifically intended to be used in the context of reliable (i.e., fault-tolerant) distributed environments design[1]. In this paper, we focus on protocol classes that involve solving the distributed consensus problem, since agreement is a central problem in many distributed algorithm that deal with failures.

### 1.1 Objects and distribution

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.

[1] We use the term "distributed system" in a very general sense, whereas we use "distributed environment" when we want to refer to set of abstractions that support the programming of distributed applications.

**Extension of object-based languages** This research stream consists in adding new specialized abstractions to object-based languages, in order to support the programming of distributed applications. The CORBA standard object framework [11] belongs to this category. The aim of this approach is to facilitate the programming of distributed applications, using high-level languages.

**Design of reflexive architectures** Research in reflexive architectures [1, 30] consists in defining basic infrastructures for describing environment architectures, in an object-based language which is also used to describe applications. Approaches that provide limited reflexive facilities, such as GARF [9] and Composition-Filters [2], can also be considered to belong to this category. The aim of reflexive architectures is to provide ways of extending distributed environments with minimal impact on applications.

**Definition of basic abstractions** A third research stream consists in defining adequate abstractions that represent distributed systems, or specific parts of distributed systems. The idea here is to structure the architecture of distributed environments in the same way it is usually done for distributed applications. Expected benefits are modularity, extensibility, flexibility, and portability of the distributed environments on which applications are built.

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 describes a research work which belongs to the third stream. We are concerned here with the design and implementation of a distributed environment as a set of objects. We focus on classes that are related to distributed agreement protocols, since the latter are fundamental to reliable distributed systems.

## 1.2 Reliability in 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* [21].

A *reliable* 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.

**Transaction paradigm** Many distributed environments provide the *transaction paradigm* [17] 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*[2] 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.

In designing the BAST class library, 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 locking), atomic commitment, etc., and are aimed at being used by system programmers, not application programmers. As we shall see, those abstractions are based on agreement protocols.

**Group paradigm** Group-oriented environments like ISIS [4] or GARF [9] 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.

Membership protocols guarantee that all members of some group $g$ agree on a totally ordered sequence of views $view_1(g)$, $view_2(g)$, ...; 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 support 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 customized.

A major characteristic of the BAST class library 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

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

adequate protocol classes. As a consequence, BAST naturally supports reliable multicasts involving different groups of replicas.

### 1.3 Protocols as structuring components

We believe that protocols should be basic structuring components of distributed environments. In the BAST class library, distributed protocols are manipulated as *classes of objects*. So, it is very easy to extend and/or customize high-level abstractions provided by distributed environments based on BAST.

**What are protocol classes?** A *protocol class* defines the behavior of objects capable of executing a particular distributed protocol. When the protocol is based on symmetric roles, only one class is necessary, while if it is based on asymmetric roles, there is the need for as many classes as they are roles involved in the protocol.

Protocol classes improve the reusability of complex algorithms, e.g., in BAST, the consensus protocol proposed by Chandra and Toueg [8] is implemented, once and for all, in reusable classes. So, customizations and optimizations are easily achieved through subclassing, and new protocols can be created and integrated to the environment with minimal efforts. This approach also provides a modular view of various distributed protocols, which helps to better understand the relationship between them. Making different protocols work together is then made easier. In BAST for example, transactions on replicated objects are achieved seamlessly because the atomic commitment protocol and the total order multicast protocol are implemented in well-defined classes, based on common *generic* protocol classes. We see the BAST library of protocol classes as our contribution to the definition of a well-structured framework for building reliable distributed environments.

**DTM generic protocol** Agreement plays a central role in many distributed algorithms that deal with failures. For this reason, classes that implement protocols solving the distributed consensus are of first importance in BAST. In the remainder of this paper, we present protocol classes that implement what we believe to be the common denominator of many reliable distributed algorithms: the *Dynamic-Terminating-Multicast* generic protocol (DTM). We also present how the corresponding protocol classes can be customized to solve the atomic commitment problem, which is central to transactional environments, and the total order multicast, which is central to group-oriented environments. Elsewhere [15], we have already proved that both a general atomic commitment and a total order multicast can be considered as instances of the DTM generic protocol.

### 1.4 About this paper

In next sections, we presents protocol classes that support the DTM generic protocol, and how those classes can be derived to build a total order multicast protocol and a general atomic commitment protocol. We also detail how

DTM classes have been implemented, and from which other protocol classes they inherit. More specifically, section 2 gives an overview of the BAST library of protocol classes, and presents its context and current status. Section 3 introduces the distributed system that we consider and the DTM generic protocol itself. Then, section 4 details the protocol classes that are given to system programmers wanting to use the DTM generic protocol; explanations on how to instantiate the generic dimensions of DTM are given here. Section 5 shows how we built a total order multicast and a general atomic commitment by subclassing DTM classes. For this section, we chose a language independent approach. The design of DTM generic protocol classes, in the context of our first Smalltalk prototype of the BAST class library, is presented in section 6. Section 7 compares the approach presented in this paper with other approaches described in the literature. Finally, section 8 summarizes what our approach, and the BAST class library that supports it, brings to reliable distributed programming, as well as the future research directions we are planning head to.

## 2 Overview of the BAST class library

The BAST library of protocol classes is implemented as the fundamental structuring component of the BASTET[3] *reliable distributed environment*. BASTET is aimed at providing a complete set of powerful abstractions, that support the design and implementation of reliable distributed applications. Figure 1 presents an overview of BASTET's architecture: apart from the operating system services (layer $h$), it is based on a fully object-oriented design and implementation. In the BASTET environment, various abstraction levels are provided, depending on the skills of programmers. At the highest level, all the complexity is hidden in ready-to-use components. This high-level layer of BASTET is an evolution of the GARF environment, which was aimed at supporting reliable distributed programming in a fairly automated way [9]. In BASTET, high-level abstractions are used to hide BAST's protocol classes to application programmers, while the BAST library is intended to be extended by system programmers.

### 2.1 BAST's protocol classes

The BAST library is based on an hierarchy of protocol classes, e.g., classes implementing objects capable of sending and receiving messages, classes implementing objects capable of detecting other objects' failures, etc. This approach allows to build protocols in an incremental way.

[3] You are probably wondering why we called our reliable distributed environment BASTET, and its protocol class library BAST. Well, *Bast*, also known as *Bastet*, was a cat-goddess in the Egyptian mythology, worshiped in the delta city of Bubastis. As you probably know, cats are said to have seven lives, which is quite a good replication rate to be fault-tolerant, isn't it! So, we thought the two names of the protectress of cats would be nice names for our reliable distributed environment and for our class library.

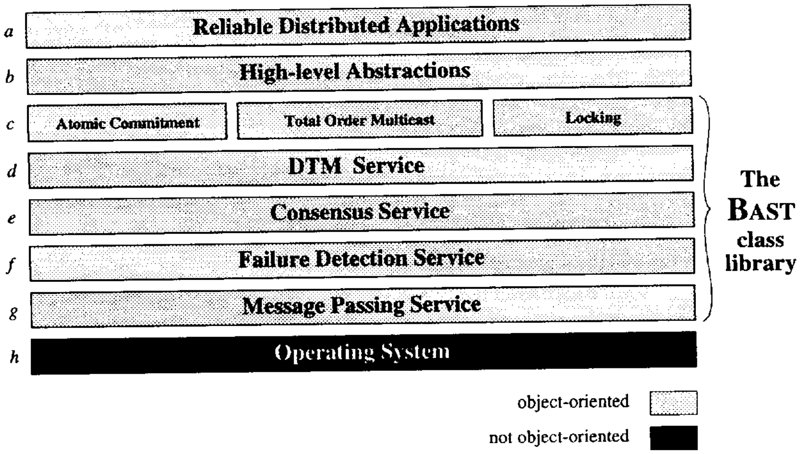

**Fig. 1.** Architecture of the BASTET reliable distributed environment

Since protocols can be manipulated as classes of objects in BAST, system programmers that have know-how in distributed systems can build new protocols, while less skilled programmers can simply use existing ones. The atomic commitment and the total order protocols, presented in section 5, are examples of such "ready-to-use" distributed protocols. New protocols can be built by expert programmers in distributed systems by instantiating generic protocols such as DTM, or by creating new ones using more basic components. Section 6 provides an overview of such lower-level components.

In the remainder of this paper, we first focus on layers $c$ and $d$ of the BAST class library. Those layers are the subjects of section 4 and section 5 respectively. Section 6 then presents layers $e$ , $f$, and $g$, and how they are used to build the DTM generic protocol (layer $d$).

## 2.2 Current status of BAST

The BAST library of protocol classes, as well as the BASTET reliable distributed environment, are developed in the context of the *Phoenix* research project, at the Operating Systems Laboratory of the Swiss Federal Institute of Technology, Lausanne. The general objective of this on-going research project is to better understand what problems are implied by reliability in distributed systems, and how they can be solved through coherent and reusable tools.

At the moment, our first implementation of the BAST class library is coming to an end, and the BASTET environment is almost fully operational. This prototype is written in Smalltalk and will be used as reference version for further developments. We have already started to work on a C++ version of the BAST library, which will be used in the *Opendreams* project (ESPRIT n°29843). This project aims at designing a CORBA compliant platform for *reliable* (distributed) industrial applications.

# 3 DTM generic protocol

## 3.1 Distributed system model

The *distributed system* we consider is composed of a finite set of *distributed objects* $\Omega = \{o_1, o_2, \ldots, o_n\}$ that communicate by message passing. The *messages passing service* provides the means to designate distant objects through *remote object references*, and to send *messages* to them; a message can be any object. Communication primitives are reliable, i.e., a message sent by some object $o_i$ to some object $o_j$ is eventually received by $o_j$, if both $o_i$ and $o_j$ do not fail. This ensured by retransmitting messages if necessary. We suppose that objects fail only by crashing and that any network failure is eventually repaired.

**Failure detectors** In this paper, we make no assumption on the synchrony of our distributed system, i.e., we are not interested in knowing if communication delays are bounded or not. Instead, we use the notion of *failure detector*. Failure detectors encapsulate the properties of the underlying distributed system, by being abstractly characterized through reliability properties (namely completeness and accuracy [8]). Failure detectors are said to be *unreliable* if they can make mistakes in incorrectly suspecting objects to be faulty. The relationship between failure detectors and distributed systems can be expressed as follow: depending on the synchrony properties of the underlying distributed system, one can build failure detectors which differ from their reliability properties. Failure detectors have been classified according to their reliability properties in [8]. Some problems need reliable failure detectors to be solved, while others only need unreliable failure detectors.

## 3.2 Generic protocol

The DTM generic protocol enables an *initiator* object to 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 by each $participant_k$ in response to $m$. Figure 2 presents an overview of the DTM generic protocol: arrows represent data exchanges, while numbers in circles show in what order data exchanges 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. From now on, we consider that objects are *a priori* on different nodes. Since the interaction of the protocol with each participant is exactly the same, arrows are numbered for $participants_i$ only.

**Why is it generic?** The DTM protocol is *generic* in the sense that the message $m$ sent by the initiator, the set of participants *Dst(m)*, the response $reply_k$ generated by each $participants_k$, and the interpretation of *Reply(m)*, the set of replies on which agreement is reached, are not defined *a priori*. One more generic dimension, the *validity condition*, allows to constrain *Reply(m)*; if that constraint

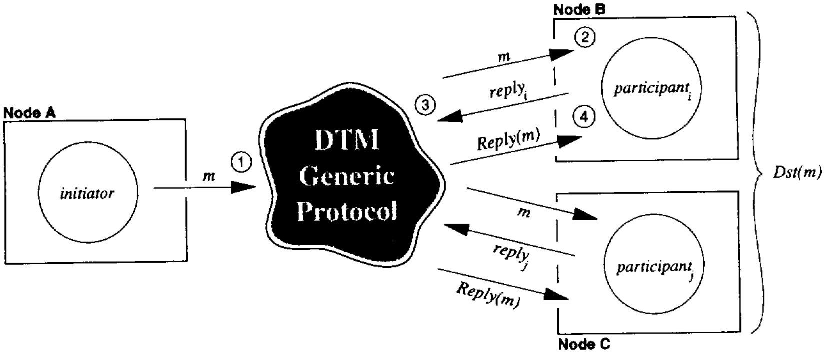

**Fig. 2.** DTM generic protocol

cannot be satisfied, the protocol will block. The reliability property of the DTM generic protocol lies in the fact that it will not *necessarily* block if one or more participants fail[4]: it depends on the chosen *validity condition*. The *Reply(m)* set received by all non-faulty participants might simply lack the replies of faulty 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. Details on how those generic dimensions can be tailored to fit the needs of specific problems are given in sections 4 and 5.

# 4 DTM generic protocol classes

## 4.1 Classes Initiator and Participant

In BAST, system programmers wanting to use the DTM generic protocol have essentially to deal with two classes, the Initiator protocol class and the Participant abstract protocol class, which are subclassed when customizing the DTM generic protocol. Instances of those subclasses will play the role of the *initiator*, respectively the *participant*, as defined in section 3.2. According to our system model, initiators and participants objects are able to perform message passing (see section 3.1). Figure 3 presents what objects and operations are involved while the protocol is executing: fat arrows picture operation invocations on objects, bullet-arrows (←•) represent objects resulting from invocations, and numbers in circles show in what order invocations occur. Not surprisingly, figure 3 is very similar to figure 2.

[4] The DTM protocol is based on (possibly unreliable) failure detectors to determine if an object is faulty or not.

**Objects executing the DTM protocol** The 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 condition*; 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 receive() operation, taking $m$ as argument. In turn, $participant_k$ computes and returns its $reply_k$. Eventually, each non-faulty participant is invoked through the interpret() operation with the $Reply(m)$ set, on which consensus has been reached, as argument. So, as long as interpret() implements a deterministic algorithm, all participants will take the same "decision". Operations receive() and interpret() are invoked through callbacks by the DTM protocol.

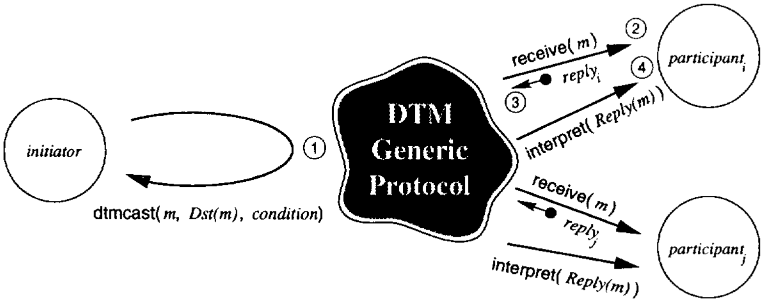

**Fig. 3.** Objects executing the DTM generic protocol

## 4.2 Generic dimensions

Since DTM is a generic protocol, one has to *instantiate*[5] it in order to solve some particular problem. To achieve this, one has to define the semantics of each *generic dimension* of the protocol. As we shall see, this can be done in many ways, depending on which dimension is considered, e.g., by deriving some class and redefining its operations, by creating an object of some class and passing it as argument to some operation involved in the protocol, etc. There are five generic dimensions to consider when instantiating the DTM generic protocol : (1) the semantics of message $m$, (2) the set $Dst(m)$ of participant objects to whom $m$ is multicast, (3) the *validity condition*, (4) the semantics of $reply_k$ generated by each participant, (5) the *interpretation* of the set of replies $Reply(m)$ on which objects in $Dst(m)$ agree.

We are now going to detail each of those generic dimensions and show how they can be used. Figure 4 presents the DTM protocol classes and their main operations, while figure 5 summarizes how generic dimensions are expressed using those classes.

[5] In this context, the verb "to instantiate" does not mean "to create an instance of some class", as in the object-oriented paradigm, but to "customize the DTM generic protocol".

**Semantics of message $m$** Class Initiator defines operation dtmcast(). The first argument of dtmcast() is an instance of some subclass of abstract class Message, which implements the generic message $m$. When instantiating DTM, one has typically to subclass the Message class to make it support the adequate semantics.

**Set $Dst(m)$ of participants** Generic dimension $Dst(m)$ (the set of participants) is implemented by the second parameter of operation dtmcast() and it is merely an instance of some Set class. This set contains remote object references, i.e., instances of class MPObjectRef provided by the *message passing service.* Those references are used in communications primitives to designate remote objects. Further details are given in section 6, which presents an overview of our first implementation, in particular by presenting the foundation classes of our message passing service. Neither Set, nor MPObjectRef are usually subclassed when instantiating DTM.

**Validity condition** The third parameter of dtmcast() is an instance of some subclass of abstract class Condition, which implements the generic *validity condition.* When subclassing Condition, one has to provide an implementation for its test() operation[6], which must yield true if the condition is satisfied and false otherwise. As for receive() and interpret(), operation test() is invoked by the protocol through a callback. This generic dimension is closely related to object failures and is at the heart of reliability issues. Its semantics and its use are detailed in section 4.3.

**Semantics of $reply_k$ and interpretation of set $Reply(m)$** Class Participant is an abstract class which declares two unimplemented operations, receive() and interpret(); it is up to subclasses of Participant to provide their implementations. Operation receive() must return an instance of some subclass of abstract class Reply, which implements the semantics of the generic $reply_k$ each participant yields when invoked by the protocol. So, to instantiate generic dimension $reply_k$, one has to subclass Reply and to provide an implementation of receive() that yields an instance of that subclass. Operation interpret() must implement the generic *interpretation* of set $Reply(m)$. The latter set is passed by the DTM protocol to each participant as an object of some Set class, through the argument of interpret().

### 4.3 Validity conditions and object failures

The validity condition is tested while the DTM protocol 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 $Dst(m)$. The latter contains objects of $Dst(m)$ that

[6] Operation test() is marked as *unimplemented* in class Condition.

| Classes | | Predefined Operations | |
|---|---|---|---|
| Initiator | | dtmcast ( *m*, *Dst(m)*, *condition* ) | |
| Participant | *(abstract)* | receive ( *m* )<br>interpret ( *Reply(m)* ) | *(unimplemented)*<br>*(unimplemented)* |
| Message | *(abstract)* | *none* | |
| Reply | *(abstract)* | *none* | |
| Condition | *(abstract)* | test ( *Dst(m)*, *Reply(m)*, *Suspect(m)* ) | *(unimplemented)* |

**Fig. 4.** Classes implementing the DTM generic protocol

are *suspected* to be faulty. Set $Suspect(m)$ is necessary for instances of the DTM generic protocol where failures have to be considered in the agreement process, i.e., when the condition on $Reply(m)$ is expressed in terms of object failures.

Examples of validity conditions are given in sections 5.1 and 5.2, which present how DTM can be used to build an atomic commitment protocol and a reliable total order multicast protocol respectively. In the atomic commitment protocol presented there, the validity condition can be expressed as follow: $\forall object_k \in Dst(m) : reply_k \notin Reply(m) \Rightarrow object_k \in Suspect(m)$. That predicate expresses the fact that the atomic commitment can only terminate when the replies of all *non-suspected* participants are in $Reply(m)$.

| Generic Dimensions | Instantiation done by ... |
|---|---|
| Semantics of *m* | subclassing class Message<br>+ passing an instance of Message's subclass to operation dtmcast() of the Initiator subclass |
| Set *Dst(m)* | building a Set of MPObjectRefs + passing it to operation dtmcast() of the Initiator subclass |
| Semantics of *reply$_k$* | subclassing class Reply + implementing operation receive() of the Participant subclass<br>+ returning an instance of Reply's subclass in operation receive() |
| Validity *condition* | subclassing class Condition + implementing its test() operation<br>+ passing an instance of Condition's subclass to operation dtmcast() of the Initiator subclass |
| Interpretation of *Reply(m)* | implementing operation interpret() of the Participant subclass |

**Fig. 5.** Instantiation of DTM generic dimensions

## 5 DTM generic protocol classes in action

We are now going to show how the generic protocol classes presented in section 4 can be customized to define higher-level abstractions for building reliable distributed environments. We will focus here on two elementary abstractions based on distributed protocols: the *reliable total order multicast* and the *atomic commitment.* As we show, both can be implemented as instances of the DTM generic protocol.

### 5.1 Atomic commitment

**Overview of the problem** 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 we still want all correct participants to agree, the problem is known as the *non-blocking* atomic commitment (NB-AC) [3]. 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 [13]. This lead to specify a weaker problem: the non-blocking *weak* atomic commitment (NB-WAC), which requires merely that no participant is *ever suspected*. Because the DTM generic protocol makes no assumption on the properties of the failure detector it uses, both the NB-AC and the NB-WAC problems can be seen as instances of DTM, depending on the failure detector considered.

**Protocol classes solving the atomic commitment** To solve the atomic commitment problem using the DTM generic protocol, Initiator is subclassed into some Transaction class and Participant into some Manager class. Class Transaction defines two new operations: begin() and end(), while class Manager implements inherited operations receive() and interpret(). When a Transaction object is created, it is initialized with a set of MPObjectRef instances, which is stored into instance variable managerSet. Those remote object references designate the managers that will be accessed during the transaction. Operations begin() and end() initiate and terminate an atomic sequence of operations respectively; both operations are invoked by the client of the transaction.

When agreement on commit or abort is reached, each manager applies the decision to the data object under its responsibility; it does so by invoking operation apply() on itself. Data objects are held in instance variable dataObject, defined by class Manager. Each Manager object also has a currentManagerSet instance variable, which contains the manager set of the transaction to which that object currently belongs[7]. Figure 6 gives an overview of the atomic commitment protocol based on DTM; as in figure 3, arrows represent invocations on objects, while numbers in circles show in what order invocations occur. In figure 7, operations and variables defined by protocol classes Transaction and Manager are presented; we do not detail secondary classes there.

We now going to sketch how objects interact while the atomic commitment protocol is executing. Figure 8 presents the implementation 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 "↑".

[7] We are not interested in the problem of concurrently accessed managers here. However, for simplicity sakes class Manager defines only one currentManagerSet, so concurrency control has to be *pessimistic*, i.e., achieved by locking managers before starting a transaction. In order to avoid dead-locks, the locking phase could be based on the reliable total order multicast protocol presented in section 5.2 (see [14] for details).

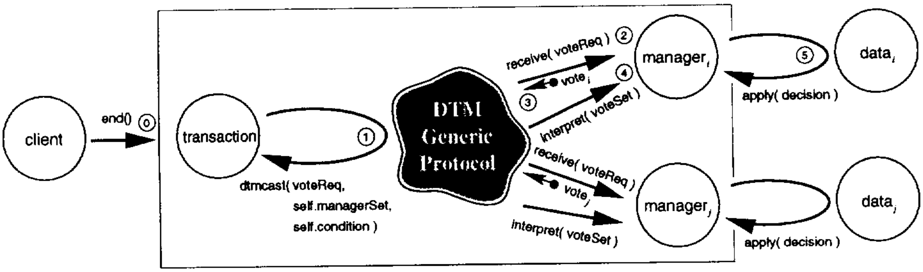

**Fig. 6.** Overview of the atomic commitment protocol with DTM

```
Class Transaction
    public operations
        begin ( managerSet );
        end ();
    instance variables
        managerSet;
        condition;
```

```
Class Manager
    public operations
        receive ( voteReq );
        interpret ( voteSet );
    private operation
        apply ();
    instance variables
        dataObject;
        currentManagerSet;
```

**Fig. 7.** Protocol classes for the atomic commitment

**On the initiator side** When a client wants to terminate an atomic sequence of operations on distinct remote objects, it invokes the end() operation on the corresponding transaction object (see figure 8 (a)). Operation end() first creates a VoteRequest message, and stores it in a local variable voteReq. Class VoteRequest inherits from class Message and defines new instance variable managerSet. After operation end() has created message voteReq, it initializes member voteReq.managerSet with the transaction's manager set[8] (self.managerSet). Then, operation end() 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 self.managerSet and self.condition. Variable self.condition contains an instance of a class ACCondition, derived from class Condition. Class ACCondition defines operation test() (see figure 8 (b)), which implements the validity condition for the atomic commitment problem. As mentioned in section 4.3, that condition is expressed in terms of manager failures. The same condition is suitable for the *NB-AC* problem and for the *NB-WAC* problem, since it all depends on the failure detector the DTM protocol is using. Predicate $\forall manager_k \in Dst(m) : reply_k \notin Reply(m) \Rightarrow manager_k \in Suspect(m)$, already discussed in section 4.3, is implemented by operation test().

[8] That set was initialized through operation begin(), by the client of the transaction.

```
end ()
  || voteReq ||
  voteReq ← VoteRequest.new();
  voteReq.managerSet ← self.managerSet;
  self.dtmcast( voteReq,
                self.managerSet,
                self.condition );
                                                  (a)
```

```
                                                  (b)
test ( managerSet, voteSet, suspectSet )
  || predicate ||
  predicate ← true ;
  foreach $manager_k \in$ managerSet do
    if $vote_k \notin$ voteSet $\wedge$ $manager_k \notin$ suspectSet then
      predicate ← false ;
  ↑ predicate;
```

```
receive ( voteReq )
  || vote ||
  self.currentManagerSet ← voteReq.managerSet;
  vote ← self.vote ( voteReq );
  ↑ vote;
                                                  (c)
```

```
                                                  (d)
interpret ( voteSet )
  || decision ||
  if | voteSet | = | self.currentManagerSet | then
    decision ← commit ;
    foreach vote $\in$ voteSet do
      if vote = no then
        decision ← abort ;
  else
    decision ← abort ;
  self.apply ( decision );
```

**Fig. 8.** Implementation of the atomic commitment

**On the participant side** When message voteReq reaches a Manager object, operation receive() first puts voteReq.managerSet in its currentManagerSet instance variable (see figure 8 (c)); it then computes its vote (*yes* or *no*) and returns it to the DTM generic protocol. The vote is an instance of class Vote, which derives from class Reply and implements generic $reply_k$. The DTM protocol then collects the votes of all non-faulty managers and put them into voteSet, a set implementing the generic *Reply(m)* set. During the collecting phase, the ACCondition object may be tested several times. Eventually, operation test() returns *true* and the DTM protocol invokes each (non-faulty) manager through operation interpret() and passes it voteSet, on which agreement has been reached. Operation interpret() then computes the final decision, which is *commit* only if all votes in voteSet are *yes* and if no manager was suspected (see figure 8 (d)). This last point is tested by Manager objects by comparing the size of voteSet with the size of their currentManagerSet; if both have the same size, it means that no manager was suspected by the DTM generic protocol. The decision is finally applied to the data object by invoking the apply() operation, which undertakes the appropriate actions.

## 5.2 Reliable total order multicast

**Overview of the problem** 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 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 explicitly specified and that set can be different for each invocation[9]. 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 [26]

The reliable total order multicast protocol that we present below has been described and proved elsewhere [26], without using the DTM generic protocol. To our knowledge, it is the first algorithm capable of solving total order multicast problem in a distributed system with *unreliable* failure detectors. Since it is quite a complex protocol, we first present it independently of its implementation as an instance of the DTM generic protocol.

**Overview of the protocol** The basic idea of the algorithm 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* [16]. 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: (1) *causal order* delivery must be ensured for all messages exchanged in the algorithm; (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; (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 (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 [4]) is necessary for the algorithm to be correct[10]. 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 (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 [26].

[9] In a broadcast primitive, the destination set is implicit and contains every object of the system, i.e., $Dst(m) = \Omega$.

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

**Protocol classes solving the reliable total order multicast** To implement the total order multicast protocol presented above, using the DTM generic protocol, we subclass both the Initiator and Participant classes. Class MulticastService is a subclass of Initiator and its instances represent the *reliable multicast service* to client objects. This service offers several reliable multicast primitives, implementing various ordering semantics, e.g., fifo order, causal order, total order. Class MulticastService defines new operation toMcast(), which implements the total order multicast primitive *TO-multicast()* defined previously. Class Replica is a subclass of Participant and it implements new operation toDeliver(), as well as inherited operations receive() and interpret(). Operations toDeliver() implements the *TO-deliver()* primitive defined above. Instances of Replica represent object replicas to the multicast service: they are in charge of computing a sequence number for each received message and reordering messages accordingly. The actual replicated server object (to which messages are finally delivered) is held in instance variable serverObject, defined by class Replica. So, when a Replica object invokes operation toDeliver() on itself, the message passed as argument is delivered to the server object, with the guaranty that total order is satisfied. Figure 9 presents an overview of the total order multicast based on the DTM generic protocol; in that figure, objects replica$_i$ and replica$_j$ are *a priori* members of different groups.

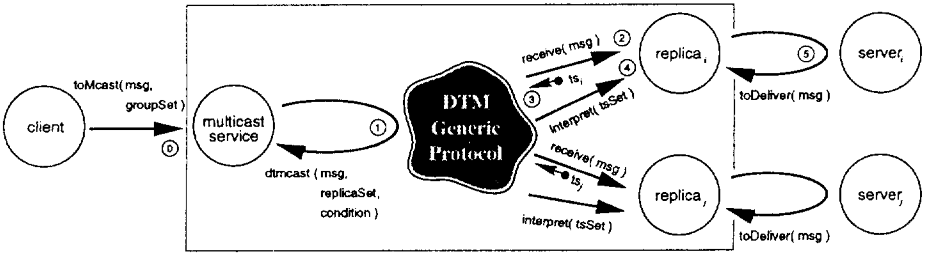

**Fig. 9.** Overview of the total order multicast protocol with DTM

We now going to sketch how objects interact while the total order multicast protocol is executing. Figure 11 presents the implementation main operations involved; the pseudo-code is the same as in figure 8

**On the initiator side** When a client object wants to issue a total order multicast to set of replicated logical objects, it first has to build a set of groups, each group containing the replicas' remote object references of one logical object. The client then invokes operation toMcast() on an instance of MulticastService, and passes it two arguments: msg, an instance of class TOMessage, and groupSet, the set of groups it just built. Operation toMcast() is implemented as follow (see figure 11 (a)): all groups of replicas are *merged* into replicaSet, a single set of remote object references; a validity condition is then created as an instance of class TOCondition; finally the invocation of dtmcast() is issued, with

```
Class MulticastService
    public operation
        toMcast ( msg, groupSet );
```

```
Class Replica
    public operations
        receive ( msg );
        interpret ( tsSet );
    private operations
        extractMsg ( tsSet );
        maxTimeStamp ( tsSet );
    instance variables
        lamportClock;
        serverObject;
        pendingQueue;
        deliveryQueue;
```

**Fig. 10.** Protocol classes for the reliable total order multicast

msg, replicaSet and condition as arguments. Class TOMessage is a subclass of Message and defines new instance variable ts; member msg.ts is used by each Replica object to hold its proposed time-stamp when msg is stored in the pending queue, and msg's sequence number when it is stored in the delivery queue. The pending queue and the delivery queue are held in Replica's instance variables pendingQueue and deliveryQueue respectively. Class TOCondition implements the validity condition of the total order multicast protocol, i.e., its test() operation evaluates the third condition presented earlier, based of the notion of qualified majority. In that condition, *Dst(m)* contains groups of objects rather than individual objects. So, TOCondition's implementation of test() simply ignores the first argument passed to it by the DTM generic protocol[11] (see figure 11 (b)). 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 TOCondition, extracts from tsSet the proposed time-stamps of a particular group and puts them in a new set.

**On the participant side** When message msg reaches a Replica object, operation receive() updates its Lamport's clock, sets msg.ts to the updated logical time and stores msg in the pendingQueue (see figure 11 (c)). It then returns msg.ts to the DTM protocol. Member msg.ts contains an instance of class TimeStamp, which derives from class Reply and implements generic $reply_k$. The DTM protocol then collects time-stamps and put them into tsSet, a set implementing the generic *Reply(m)*. During the collecting phase, the TOCondition object may be tested several times. Eventually, operation test() returns *true* and the DTM protocol invokes each (non-faulty) replica through operation interpret() and passes it tsSet on which agreement has been reached. Operation interpret() then computes msg's sequence number, moves msg from the pendingQueue to the deliveryQueue, and performs toDeliver() for all messages that have been made *deliverable* by the newly computed sequence number  (see figure 11 (d)). Operation interpret()

[11] First argument of test() contains the set of participants to the DTM generic protocol, i.e., Replica objects in that case, not groups.

```
toMcast ( msg, groupSet )
  || replicaSet condition ||
  replicaSet ← $\bigcup_{group \in groupSet} group$
  condition ← TOCondition.new();
  condition.groupSet ← groupSet;
  self.dtmcast( msg,
                replicaSet,
                condition );
```

(a)

(b)

```
test ( replicaSet, tsSet, suspectSet )
  || predicate ||
  predicate ← true ;
  foreach group ∈ self.groupSet do
    if | self.select( tsSet, group ) | ≤ ½ × | group | then
      predicate ← false ;
  ↑ predicate;
```

```
receive ( msg )
  || ts ||
  ts ← self.lamportClock.update( msg );
  msg.ts ← ts;
  self.pendingQueue.add( msg );
  ↑ ts
```

(c)

(d)

```
interpret ( tsSet )
  || msg ||
  msg ← self.extractMsg( tsSet );
  msg.ts ← self.maxTimeStamp( tsSet );
  self.pendingQueue.remove( msg );
  self.deliveryQueue.add( msg );
  foreach $m_d$ ∈ self.deliveryQueue do
    if $\forall m_p$ ∈ self.pendingQueue : $m_p$.ts > $m_d$.ts then
      self.toDeliver( $m_d$ );
      self.deliveryQueue.remove( $m_d$ );
```

**Fig. 11.** Implementation of the reliable total order multicast

relies on private operations extractMsg() and maxTimeStamp(), defined by class Replica. Those two operations allow to get the message associated to tsSet, and to compute the maximum time-stamp in tsSet respectively.

# 6 Design and implementation

## 6.1 Current prototype of the BAST class library

Our current prototype of the BAST library of protocol classes was implemented using the *Smalltalk* language and environment [10]. More specifically, we used VisualWorks, the commercial Smalltalk platform by ParcPlace Systems, Inc. The development took place on a network of Sun SPARCstations running the Solaris 2.4 operating system. Everything is an object in BAST, and Smalltalk was well-suited to support such an approach. As shown in figure 1, BAST is based on a *layered architecture* and provides various services, each of which corresponds to a particular protocol.

## 6.2 Inside the DTM generic protocol

In previous sections, we have presented the *public interfaces* of the DTM protocol classes, and how to use them. We are now going to "open" the DTM generic protocol, by presenting the "hidden face" of classes Initiator and Participant, i.e., how they implement DTM and on what other protocol classes they rely to do this.

The nature of the DTM protocol suggests that Participant objects have to be able to solve the *distributed consensus problem*, since they have to reach an agreement on the generic *Reply*($m$) (interpreted at the end of the protocol).

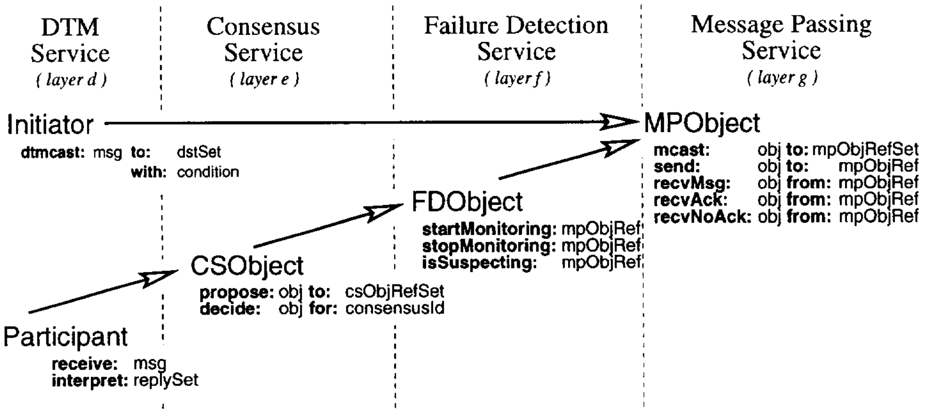
DTM Service ( layer d )
Consensus Service ( layer e )
Failure Detection Service ( layer f )
Message Passing Service ( layer g )
Initiator
dtmcast: msg to: dstSet
with: condition
MPObject
mcast: obj to: mpObjRefSet
send: obj to: mpObjRef
recvMsg: obj from: mpObjRef
recvAck: obj from: mpObjRef
recvNoAck: obj from: mpObjRef
FDObject
startMonitoring: mpObjRef
stopMonitoring: mpObjRef
isSuspecting: mpObjRef
CSObject
propose: obj to: csObjRefSet
decide: obj for: consensusId
Participant
receive: msg
interpret: replySet

**Fig. 12.** BAST's protocol class hierarchy

Furthermore, if Participant objects were not able to detect (reliably or not) their faulty peers, the failure of a single participant would lead the protocol to block. This suggests that Participant objects must be able to monitor and to *suspect* each others. Finally, while the DTM generic protocol is executing, distant objects exchange messages, so Initiator and Participant objects should be capable of performing *message passing.*

Those considerations lead us to design the *class hierarchy* presented in figure 12, where arrows represent "*is subclass of*" relationships. In this hierarchy, classes are *protocol classes* as defined previously, that is, they implement objects that are capable of executing a particular protocol. Apart from DTM, all services of figure 12 are based on protocols with *symmetric roles*, so a single class is defined *per protocol.* In this figure, only *new* operations defined by each class are represented, e.g., class FDObject refines operation recvAck:from: but we only placed that operation under class MPObject. Private/secondary operations and classes are not represented. Each service in figure 12 corresponds to a layer in figure 1. Below, we detail classes that support the protocols corresponding to those services; the presentation order follows a bottom-up approach. Figures 13 and 14 present the interfaces of those protocol classes; for each class, we list newly defined operations, as well as inherited ones that are refined.

**Message passing service** Smalltalk does not provide support for distributed objects, so we implemented the foundation for distant objects to be able to send messages to each others: *the message passing service.* This service is the base of all other distributed protocols and defines two classes: MPObject and MPObjectRef. Instances of MPObject represent *remote objects* that are capable of executing *the message passing protocol*, i.e., they know how send (and receive) any object obj to (from) each others. Sending achieved by executing either operation send:to: or mcast:to:, while receiving in performed through a callback to recvMsg:from: by the message passing service. Because communications are *reliable*, callbacks to operations recvAck:from: and recvNoAck:from: are also per-

```
Class MPObject
  instance operations
    send:to:
    mcast:to:
    recvMsg:from:
    recvAck:from:
    recvNoAck:from:
  class operations
    dispatchMsg:to:
    dispatchAck:to:
    dispatchNoAck:to:
```

```
Class FDObject
  instance operations
    startMonitoring:
    stopMonitoring:
    isSuspecting:
    recvMsg:from:
    recvAck:from:
    recvNoAck:from:
  class operations
    dispatchMsg:to:
    dispatchAck:to:
    dispatchNoAck:to:
  instance variable
    suspectSet
```

```
Class CSObject
  instance operations
    propose:to:
    decide:for:
    recvMsg:from:
    recvAck:from:
    recvNoAck:from:
  class operations
    dispatchMsg:to:
  instance variable
    currentConsensusSet
```

**Fig. 13.** Protocol classes MPObject, FDObject and CSObject

formed by the protocol when necessary; class MPObject implements those operations so they do nothing. Instances of MPObjectRef are used to designate remote objects and are passed as arguments to communication operations.

Reliable communications are implemented through *timeouts* and possible *retransmissions*. This task is managed by *private* class MessagePassing, which represent the message passing service on each node[12]; callbacks on MPObject are triggered by this class. To implement reliable communications, class MessagePassing relies on a low-level layer written in C: *the Reliable Communication Layer*[13] (*RCL*), developed in the context of the *Phoenix* project. This layer is built on top of the UDP protocol and adds reliability to it. This approach has the advantage to improve performance, since low-level message management, such as message packing and retransmitting, duplicated messages filtering, etc., is done in C. Furthermore, we are currently integrating RCL into Solaris 2.4 kernel, which should make the communication even faster.

On each node, class MessagePassing opens a UDP port used to send (and receive) messages to (from) others nodes. The mapping between local MPObject instances and the node where they can be found is managed by the MessagePassing class on that node. However, the dispatching of incoming messages is delegated by class MessagePassing to the class of the target object, e.g, if a message is addressed to some FDObject, then it is up to class FDObject to dispatch it. In order to do this, class MPObject defines *class operations* dispatchMsg:to:, dispatchAck:to: and dispatchNoAck:to:, which can be redefined by its subclasses. As we shall see in next paragraph, this mechanism is useful for example when common actions have to be undertaken for all instances of some subclass of MPObject, each time a message arrives on the node. Class MessagePassing is

[12] As said in section 3, different nodes imply different *address spaces*. In Smalltalk, this means that a node corresponds to an execution of the *Smalltalk virtual machine*.

[13] The VisualWorks environment provides powerful facilities that enable to *dynamically link* C libraries to the Smalltalk virtual machine, and then to call C functions from those libraries within Smalltalk code.

also responsible for marshaling and unmarshaling objects sent through RCL; this is done by using the *Binary Object Storage Service* (*BOSS*) provided by the VisualWorks environment.

**Failure detection service** In our approach, object failures are dealt with by *failure detectors*, and in BAST, this concept is supported by the *failure detection service.* This service is based on FDObject, a subclass of MPObject which implements the behavior of objects capable of executing *the failure detection protocol.* Each instance of FDObject manages suspectSet, a private set containing the MPObjectRef of every remote object it is currently suspecting. Class FDObject redefines operations recvMsg:from:, recvAck:from: and recvNoAck:from: so they adequately add or remove the MPObjectRef passed to them by the failure detection protocol. This class also implements new operations startMonitoring:, stopMonitoring: and isSuspecting:. The first two operations can be used to start and stop "pinging" a remote object respectively. Operation isSuspecting: returns true if the invoked FDObject is currently suspecting the remote object which reference is passed as argument.

In our first prototype of BAST, we made the assumption that objects on the same node do not fail *separately*, i.e., we only consider failures of Smalltalk virtual machines, not of individual objects. This is fairly reasonable, since we only consider *crash failures*: it makes little sense having one object to crash while others on the same node (i.e., on the same address space) don't. As a consequence, whenever a message is received on node $A$ from a remote object located on node $B$, all objects on node $A$ that are suspecting some object located on node $B$ should stop doing so. Similarly, when an object on node $B$ is no more reachable by some object on node $A$, all objects on node $A$ that interested in some object located on node $B$ should add it in their private suspectSet. The dispatching of suspicion additions and removals is performed by class FDObject, through its class operations dispatchMsg:to:, dispatchAck:to: and dispatchNoAck:to:.

**Consensus service** Agreement is a central problem of many reliable distributed protocols, in particular all those that can be viewed as instances of the DTM generic protocol. In BAST, support for solving the agreement problem is provided by *the consensus service*, which implements the $\diamond S$-consensus protocol proposed by Chandra and Toueg [8]. That protocol enables objects to reach agreement despite failures, in asynchronous systems augmented with *unreliable* failure detectors of class $\diamond S$ (see [8] for further details on the classification of failure detectors).

Our consensus service relies on CSObject, a subclass of FDObject which implements the behavior of objects that can execute the aforesaid consensus protocol. In the $\diamond S$-consensus protocol, objects executing the consensus protocol must be able to *suspect* each others; this is why a CSObject is also an FDObject. When a consensus protocol begins, each CSObject involved in it starts monitoring its peers, by calling operation startMonitoring: inherited from class FDObject.

A CSObject $o_i$ stops monitoring another CSObject $o_j$, as soon as all executions of the consensus protocol involving both $o_i$ and $o_j$ terminate. This is done by invoking operation stopMonitoring:, inherited from class FDObject.

Instances of CSObject are capable of running more than one execution of the consensus protocol simultaneously, and they can propose *any* object as agreement value. Class CSObject defines new operations propose:to: and decide:for:. When a CSObject is invoked through operation propose:to:, a new execution of the consensus protocol starts for that object. As first argument, operation propose:to: takes *any* object and proposes it as its agreement value to all consensus objects referenced in its second argument, set csObjRefSet. Internally, a *unique* identifier is associated to that particular execution of the consensus. When an agreement is reached, class CSObject on each non-faulty node eventually receives the decision message and has to *dispatch* it to all local object(s) involved in that execution of the protocol. Instead of calling operation recvMsg:from: on the concerned object(s), operation dispatchMsg:to: performs a callback to decide:for: on them. First argument of decide:for: is the object on which agreement has been reached, while second argument is the identifier associated to that execution of the consensus protocol.

```
Class Initiator
    instance operation
        dtmcast:to:with:
```

```
Class Participant
    instance operations                 instance operations (continued)
        dtmProtocolFor:                     collecReplies
        storeDstSetOf:as:                   stopCollecReplies
        getDstSetOf:                        waitForDecision
        storeCollectSetOf:as:               dtmIdsConcernedBy:
        getCollectSetOf:                    recvMsg:from:
        storeValidityCondOf:as:             recvAck:from:
        getValidityCondOf:                  recvNoAck:from:
```

**Fig. 14.** Protocol classes Initiator and Participant

**DTM service** We are now ready to look at the implementation of DTM protocol classes Initiator and Participant. Those classes mostly rely on inherited operations from their superclasses. Initiator and Participant's interfaces are summarized in figure 14, while figure 15 presents the Smalltalk code of the three main operations involved in the DTM generic protocol, i.e., dtmProtocolFor:, recvMsg:from:, and recvNoAck:from:. Those operations are defined by class Participant.

Class Initiator derives from MPObject, the class at the top of our hierarchy. Its implementation of operation dtmcast:to:with: directly relies on inherited operation mcast:to:. When invoked, operation dtmcast:to:with: initializes *private fields* of its first argument msg, an instance of a Message's subclass. It then multicasts msg to dstSet, the set of participants passed to it as second argument. Class

Message defines three private fields: dtmId, an identifier used to distinguish different execution of the DTM protocol, vCond, the validity condition, and dstSet, the set of participants to the protocol. Those informations are initialized using arguments passed to dtmcast:to:with:.

```
recvMsg: obj from: mpObjRef
    "Let super-classes do their job first"
    super recvMsg: obj from: mpObjRef.
    ...
    "Treat incoming DTM messages"
    (obj isKindOf: Message)
        ifTrue:
        [
            self dtmProtocolFor: obj.
        ]
    ...
                                                    (a)
```

```
dtmProtocolFor: msg
    | reply collectSet replySet |

    "Store information related to that new execution"
    self storeDstSetOf: (msg dtmId) as: (msg dstSet).
    self storeValidityCondOf: (msg dtmId) as: (msg vCond).

    "Execute DTM generic protocol phases"
    reply := self receive: msg.
    self mcast: reply to: (msg dstSet).
    collectSet := self collectReplies.
    self propose: collectSet to: (msg dstSet).
    replySet := self waitForDecision.
    self interpret: replySet
                                                    (b)
```

```
recvMsg: obj from: mpObjRef                         (c)
    "Let super-classes do their job first"
    super recvMsg: obj from: mpObjRef.
    ...
    "Treat incoming DTM replies"
    (obj isKindOf: Reply)
        ifTrue:
        [ | dstSet collectSet vCond ready |

            "Get information related to that execution"
            dstSet := self getDstSetOf: (obj dtmId).
            collectSet := self getCollectSetOf: (obj dtmId).
            vCond := self getValidityCondOf: (obj dtmId).

            "Add reply and test validity condition"
            collectSet add: (obj replyValue).
            ready := vCond
                        testWith: dstSet
                        with: collectSet
                        with: suspectSet.

            "Possibly stop collecting replies"
            ready
              ifTrue:
              [
                 self stopCollectReplies.
              ]
        ]
    ...
```

```
recvNoAck: obj from: mpObjRef                       (d)
    | dstSet collectSet vCond ready |

    "Let super-classes do their job first"
    super recvNoAck: obj from: mpObjRef.
    ...
    "For each current execution of DTM, do..."
    (self dtmIdsConcernedBy: obj)
      do:
      [ :dtmId |

        "Get information related to that execution"
        dstSet := self getDstSetOf: dtmId.
        collectSet := self getCollectSetOf: dtmId.
        vCond := self getValidityCondOf: dtmId.

        "Test validity condition"
        ready := vCond
                    testWith: dstSet
                    with: collectSet
                    with: suspectSet.

        "Possibly stop collecting replies"
        ready
          ifTrue:
          [
            self stopCollectReplies.
          ]
      ]
    ...
```

**Fig. 15.** Smalltalk implementation of the DTM generic protocol

Since DTM involves the execution of the consensus protocol, Participant is a subclass of CSObject. When msg reaches a node, the local message passing service invokes the target participant through operation recvMsg:from:. The participant then invokes operation dtmProtocolFor: on itself, passing it msg (see figure 15 (a)). This call starts the DTM generic protocol, which has its various phases implemented through operation invocations in dtmProtocolFor: (see figure 15 (b)). The latter starts by storing informations related to that new execution of DTM and then invokes operation receive: with argument msg. Remember that operation receive: is implemented differently by each subclass of Participant, and that it returns reply, an instance of some Reply's subclass. Object reply is then sent to all objects referenced in msg's dstSet. Operation collectReplies is then invoked, and blocks until the validity condition associated to msg is satisfied. This condition is re-tested each time a *reply* or a *negative acknowledgment* (no ack) is received by the participant (see figures 15 (c) and (d)). There might be more than one execution of the DTM protocol that is concerned by an incoming *no ack*. Operation dtmIdsConcernedBy: returns a set containing the dtmId of each execution that should have its validity condition re-tested.

Eventually, the call to collectReplies returns and operation propose:to: is invoked with initSet, the set of collected replies, and msg's dstSet as arguments. The invocation of operation propose:to:, inherited from class CSObject, starts the $\diamond\mathcal{S}$-consensus protocol; this call is non-blocking. By invoking waitForDecision, operation dtmProtocolFor: blocks until the consensus terminates. When operation waitForDecision returns, it yields replySet, the set of replies on which agreement has been reached. Finally, the protocol invokes operation interpret:, which implementation is delegated to subclasses of Participant, and passes it replySet as argument. This call concludes the execution of the DTM generic protocol. In figure 15 (b), we emphasized the two operations that enable to customize the DTM generic protocol, i.e., operations receive: and interpret:.

# 7 Related work

## 7.1 Distributed programming abstractions

Although not applied to the same research domain, our approach is similar to those of [18, 6, 7], in that the main objective is to define the basic generic abstractions for building a modular distributed environment. In the BAST library of protocol classes, we are concerned with building distributed agreement protocols, while [18] focuses on concurrent programming and [6, 7] focus on persistence storage and distributed object communication and execution.

O.L Madsen has presented in [18] a library of classes representing high-level abstractions for concurrent programming, such as *Rendez-Vous* and *Monitors*. Those abstractions are built on top of the lower level *Semaphore* abstraction. Following the same approach, BETA, described by S. Brandt and O.L Madsen in [6], is a set of "mandatory" abstractions to support distributed object execution, and remote object invocation. Among these abstractions are: *Ensemble*, representing a physical network node, *Shell*, representing a self-contained program module, and *NameServer*, providing a mapping between textual names and object references. CHOICES [7] is an example of a class library representing a distributed operating system. Traditional elements of operating systems, such as *Process*, *Domain* or *Disk*, are implemented as classes with well-defined interfaces. One can then customize the operating system, through inheritance, in order to match particular application needs.

### 7.2 Transactional libraries

D. McCue has presented a class library which enables to attach persistence and transactional features to application objects [19]. An interesting aspect of the library is the orthogonality of characteristics: (1) an object is persistent if its class inherits from class *Persistent*, (2) an object has a dedicated concurrency control if its class inherits from class *Concurrency-Controlled*, and (3) an object is recoverable if its class inherits from class *Recoverable* [29, 5]. These characteristics can be obtained separately but also together, through multiple inheritance. So, one can design objects with all transactional characteristics: serialisability, failure atomicity and permanence.

In [22, 24, 12], the transactional system itself is designed as a class library. This leads to a better modularity, and has enabled to change the underlying transactional protocols with minimal effects on the rest of the system. In [22], a (pessimistic) two-phases locking protocol can be customized for each class according to its semantics. In [24, 12], pessimistic concurrency control protocols are replaced by optimistic ones. All these changes are done with no effects on application objects.

However, none of the research mentioned above discuss the way transactional protocols, such as distributed locking and distributed atomic commitment, can be designed and implemented on top of lower level reusable components. With those approaches, protocols are assumed to exist and to be provided by the underlying distributed environment. Our work can be viewed as complementary to these research works. Our approach precisely aims at providing a generic way to design and implement protocols, particularly *agreement* protocols (e.g., for transactions), from more basic components, such as DTM protocol classes. The BAST class library neither addresses how to attach transactional features to application objects, nor is concerned with the way these protocols can be composed to build a transactional system.

### 7.3 Communication protocol libraries

Several authors have discussed the need for designing and implementing libraries of communication protocols. The STREAMS framework [25], a pioneer in the domain, and the $x$-Kernel [23], 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 [28] and the CONSUL [20] 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. As we have discussed in section 1, transaction-oriented applications are very difficult to support on top of such group-oriented systems. D.C. Schmidt introduced the ASX framework [27], a set of C++ components that help building reusable communication infrastructures. Those components, also known as *wrappers*, are aimed at performing common communication-related tasks, e.g., connection establishment, routing, etc. However, there is no such thing as protocol classes in ASX, which can be seen as a toolbox of reusable components.

## 8 Concluding remarks

In this paper, we have introduced BAST, a library which offers a coherent hierarchy of *protocol classes.* Protocol classes are aimed at helping *system* programmers in building abstractions provided to *application* programmers. They can be used as the basic structuring components of distributed environments, and they significantly improve modularity. They also facilitate the customization and optimization of existing protocols, and enables to create new protocols very easily by subclassing.

In fault-tolerant distributed environments, protocol that enable to reach agreement despite failures play an essential role. The BAST library provides protocol classes that implement *DTM*, a generic protocol that can be customized to solve problems in distributed systems where failures can occur. We have presented how DTM generic protocol classes can be derived to solve the *atomic commitment problem*, and the *reliable total order problem.*

There are many research works on how to design distributed environments in terms of objects, but protocols are usually not modeled as classes. STREAMS [25] and *x*-Kernel [23] provide libraries of communication protocols but do not address fault-tolerance, while HORUS [28] and CONSUL [20] do. However, none of those systems view protocols as classes: protocols are dealt with as sets of functions. BAST is the only library of *protocol classes* we know of that addresses reliability issues. Furthermore, it provides protocol classes that support both the transaction paradigm and the group paradigm; this allows to smoothly integrate transactions on replicated objects. We see BAST as our contribution to the design of well-structured *fault-tolerant* distributed environments.

Our first prototype of the BAST class library is implemented in Smalltalk. We are currently implementing a C++ version of BAST, which will be used as base for other research projects. Future work will also consist in studying other protocols that are used to achieve fault-tolerance in distributed systems, and in seeing how they can fit into the BAST class hierarchy.

## References

1. G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III Proceedings (DCCA-3)*, pages 197–207. IFIP Transactions, 1993. Elsevier.
2. M. Aksit, K. Wakita, J. Bosh, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Object-Based Distributed Programming,*

volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer Verlag, 1993.

3. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.
4. K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.
5. A. Black. Understanding transactions in the operating system context. *Operating Systems Review*, 25(28):73–77, January 1991.
6. S. Brandt and O.L Madsen. Object-oriented distributed programming in Beta. In *Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 185–212. Springer Verlag, 1993.
7. 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.
8. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR94-1458, Cornell University, Computer Science Department, October 1994. A preliminary version appears in PODC'91.
9. B. Garbinato, R. Guerraoui, and K.R. Mazouni. Implementation of the GARF replicated object plateform. *Distributed Systems Engineering Journal*, 2:14–27, 1995.
10. A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley, 1983.
11. Object Management Group and X/Open. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 1990. Document No. 91.12.1 (Revision 1.1).
12. R. Guerraoui. Modular atomic objects. *Theory and Practice of Object Systems*, 1(2):89–100, 1995.
13. 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.
14. R. Guerraoui and A. Schiper. A generic multicast primitive to support transactions on replicated objects in distributed systems. In *IEEE International Workshop on Future Trends in Distributed Computing Systems (FTDCS-95)*, August 1995. Korea.
15. 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.
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
17. N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Koffmann, 1994.
18. O.L Madsen. Building abstractions for object-oriented programming. Technical report, University of Arhus, Computer Science Department, February 1993.
19. D. McCue. Developing a class hierarchy for object-oriented transaction processing. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, pages 413–426, Utrecht (Netherland), June/July 1992. Springer Verlag.

20. S. Mishra, L. Peterson, and R. Schlichting. Experience with modularity in Consul. *Software-Practice and Experience*, 23(10):1053–1075, October 1993.
21. S. Mullender, editor. *Distributed Systems*. ACM Press, 1989.
22. 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.
23. L. Peterson, N. Hutchinson, S. O'Malley, and M. Abott. Rpc in the $x-$Kernel: Evaluating new design techniques. *ACM Symposium on Operating Systems Principles*, 23(10):91–101, November 1989.
24. 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.
25. D. Ritchie. A stream input-output system. *Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.
26. 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.
27. D.C. Schmidt. ASX: an object-oriented framework for developing distributed applications. In *Proceedings of the 6th USENIX C++ Technical Conference*. USENIX Association, April 1994.
28. R. van Renesse and K. Birman. Protocol composition in Horus. *ACM Principles of Distributed Computing*, 1995.
29. J. Wing. Decomposing and recomposing transaction concepts. In *Workshop OBDP93*, pages 111–122, 1994.
30. Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'92)*, pages 414–434. ACM Press, October 1992. Special Issue of Sigplan Notices.