Effective multicast programming in large scale distributed systems

Patrick Th. Eugster[1,*,†], Romain Boichat[1], Rachid Guerraoui[1], Joe Sventek[2]

[1] *Swiss Federal Institute of Technology, Lausanne*
[2] *Agilent Laboratories Scotland, Edinburgh*

SUMMARY

Many distributed applications have a strong requirement for efficient dissemination of large amounts of information to widely spread consumers in large networks. These include applications in e-commerce and telecommunication. Publish/subscribe is considered one of the most important interaction styles to model communication at large scale. Producers publish information for a topic and consumers subscribe to the topics they wish to be informed of. The decoupling of producers and consumers in time, space, and flow makes the publish/subscribe paradigm very attractive for large scale distribution, especially in environments like the Internet.

This paper describes the architecture and implementation of DACE (Distributed Asynchronous Computing Environment), a framework for publish/subscribe communication based on an object-oriented programming abstraction in the form of Distributed Asynchronous Collection (DAC). DACs capture the different variations of publish/subscribe, without blurring their respective advantages. The architecture we present is tolerant to network partitions and crash failures. The underlying model is based on the notion of Topic Membership: a weak membership for the parties involved in a topic. We present how Topic Membership enables the realization of a robust and efficient reliable multicast for large scale. The protocol ensures that, inside a topic, even a subscriber that is temporarily partitioned away eventually receives a published message.

KEY WORDS: *Concurrency, scalability, reliability, multicast, membership, partitions*

---

*Correspondence to: Patrick Th. Eugster, Distributed Programming Group, Communication Systems Department (DSC), Federal Institute of Technology Lausanne (EPFL), Switzerland, CH-1015
Contract/grant sponsor: This work is partially supported by Agilent Laboratories and Lombard Odier & Co.
†E-mail: Patrick.Eugster@epfl.ch

1. INTRODUCTION

This paper presents the multicast capabilities of *DACE (Distributed Asynchronous Computing Environment)*: a middleware solution based on *publish/subscribe* interaction schemes. In particular, this paper focuses on how *DACE* enables the efficient and reliable multicast of information at large scale, despite process and network failures.

1.1. Motivation

Most research efforts in the context of distributed computing is either undertaken to find protocols for various reliability requirements [39], or to develop more easy-to-use programming abstractions for remote interaction [5]. The multitude of existing multicast protocols for various system and failure models are very good examples of the first class. The second research axis has brought out, within others, derivatives of the commonly employed *remote procedure call (RPC)*: middleware packages, like CORBA [35], DCOM [30] and Java RMI [44], seem to show the path for the future of practical distributed computing. However, remote object invocations are intuitive but tie applications to rigid client/server-like interactions. On the other hand, protocols developed without programming models in mind lead to low-level service implementations which are very cumbersome to use. We present in this paper an approach where the programming abstractions are tailored to reflect the underlying protocols, and conversely these protocols have been designed with a clear vision of the programming abstraction that will encapsulate them.

1.2. Communication model and programming abstraction

The most popular programming abstraction for distributed computing nowadays is the remote procedure call. The success of object-oriented middleware solutions originates from the relatively short learning phase which enables them to be put to work quickly. However, derivatives of the remote procedure call communication model present two major drawbacks. First, they do not address the increasing demand for *one-to-many* invocation semantics. Multicast and broadcast mechanisms have been a topic of intense research and development for many years. A recent study [28] shows that 30 percent of internet traffic is multicast and forsees a growth up to 50 percent in the next few years. Second, solutions based on the remote method invocation model try to hide distribution, which is both dangerous and misleading, since distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction, especially in the presence of network or component failures [21].

The *publish/subscribe* interaction style has proven its ability to overcome these shortcomings [37]. In contrast to the remote procedure call paradigm, it does not force synchronization between information producers and consumers; the participants are anonymous with respect to each other, i.e., they do not have to be known whether by *number* nor by *identity* or *location.* The participants are therefore *decoupled* in *time*, *space* as well as in *flow*, and this threefold decoupling represents a key to scalability (time decoupling: the interacting parties do not need to be up at the same time; space decoupling: the interacting parties do not need to know each other; flow decoupling: information sending/receiving does not block the main thread of control).

There are different established variants of the publish/subscribe interaction model, each one presenting its respective advantages as well as shortcomings. The classical *topic-based* or *subject-based* style involves a static classification of the messages by introducing group-like notions [39], and is incorporated by most industrial strength solutions, e.g., [9, 45]. A more recent alternative is *content-based* (*property-based* [42]) publish/subscribe [10, 43, 2]. The latter removes entirely the “arbitrary” division of the message space, and lets consumers delineate their individual interests by expressing *properties* of messages they wish to receive. However it introduces an important overhead due to matching of the messages with the subscribers criteria. In [15], we furthermore introduce a new variant, called *type-based*, which uses a classification of message *objects* based on their *type*. These alternatives are very promising and still being explored.[†]

Instead of emphasizing their differences, we bring all these variants to a common denominator. To capture the variants of publish/subscribe, we propose a high-level abstraction called *Distributed Asynchronous Collection (DAC)*. A *DAC* differs from a conventional collection by its distributed nature and the way objects interact with it: besides representing a collection of objects (*set*, *bag*, *queue*, etc.), a *DAC* can be viewed as a publish/subscribe engine of its own. In fact, when querying a *DAC* for objects fulfilling certain conditions, the client expresses its interest in such objects. In other words, the invocation of an operation on a *DAC* expresses the notion of *future notifications* and can be viewed as a *subscription*. The *DAC* abstraction enables the unification of different publish/subscribe styles in a single framework. The *Distributed Asynchronous Computing Environment (DACE)* can be seen as an extension of a conventional collection framework, like *JGL* [34]. It is composed of a hierarchy of *DAC* interfaces and classes, spanning multiple publish/subscribe variants and qualities of service. In this paper we describe the protocols underlying the implementation of a *DACE* sample class, which guarantees reliable delivery of events to all subscribers in spite of failures.

1.3. System and failure model

The protocols we use in *DACE* have been designed specifically to meet the properties of our *DAC* programming abstraction, which means that they are targeted at large scale applications. In that context, partitionings (in the context of this paper, we define *partitioning* as the creation of at least two *partitions*, while a *partition* is a subset of the participating processes) of the communication network is an extremely important aspect. It might result in service *degradation* but it should not affect the *liveness* of an application. There are several partition models in distributed group communication, like the *primary-partition* model (e.g., [6]), where only processes in the partition that contains a majority of processes are allowed to make progress. With the *minority-partition* or *partitionable* model (e.g., [26]), processes in multiple

[†]For brevity, these styles are not presented in detail in this paper.

partitions progress even if they receive only a subset of the messages, increasing the availability of the system.

In the context of this paper, we focus on a new failure model made-to-measure for the strongly decoupled nature of publish/subscribe. It tolerates crash failures as well as partitionings, and does not rely on a strongly consistent *view* shared by members, but achieves its goal through an exchange of views that is strongly self-stabilizing in a sense similar to the notion of self-stabilizing systems defined by Dijkstra [12]. The approach is comparable to *anti-entropy* protocols [19, 40]. It is less restrictive than the *majority-partition* and *minority-partition* models that rely on consensus, and requires less application support than the *partition-aware* [4] model.

The *Topic Membership* protocol we present in this paper coordinates the local views of participants of a topic in two phases. During the *stabilization* phase, participants exchange their views. Eventually, they converge to the same view. Then, the participants are in a *stabilized* phase. With the stabilization property and with partition information sharing, we are able to realize a reliable broadcast in partitions on top of *Topic Membership*.

The reliable broadcast protocol for topic-based publish/subscribe called *Topic Broadcast* that we present as an example, ensures that every subscriber eventually receives a message even if the publisher or the subscriber, itself, has crashed or has been partitioned away temporarily.[‡] In the *stabilized* phase, the protocol uses partition information to efficiently route messages. During the *stabilization* phase, the protocol enables the sending of messages, although these might not be delivered in an optimal manner.

1.4. Roadmap

The remainder of this paper is organized as follows. Section 2 gives an overview of our publish/subscribe system focusing on topic-based publish/subscribe. Section 3 presents the *DACE* framework and the underlying *DAC* programming abstraction. The system and failure model we adopt are outlined in Section 4, which allows us to formally specify the lightweight topic membership used in *DACE* in Section 5. As an example Section 6 illustrates our reliable broadcast based on TopicMembership. In Section 7 we outline the implementation of our framework and discuss some performance issues, and Section 8 contrasts our efforts with related work. Finally Section 9 summarizes our work and concludes the paper.

2. OVERVIEW OF DACE

This section gives a general overview of our *DACE* framework for large scale communication. *DACE* can be seen as a message-oriented middleware solution. It is inherently object-oriented, and is used as a lightweight library. The different layers are shown in Figure 1 and introduced

[‡]Of course this is only provided if the publisher crashes after it finished publishing the message and the subscriber eventually recovers.

Prepared using cpeauth.cls

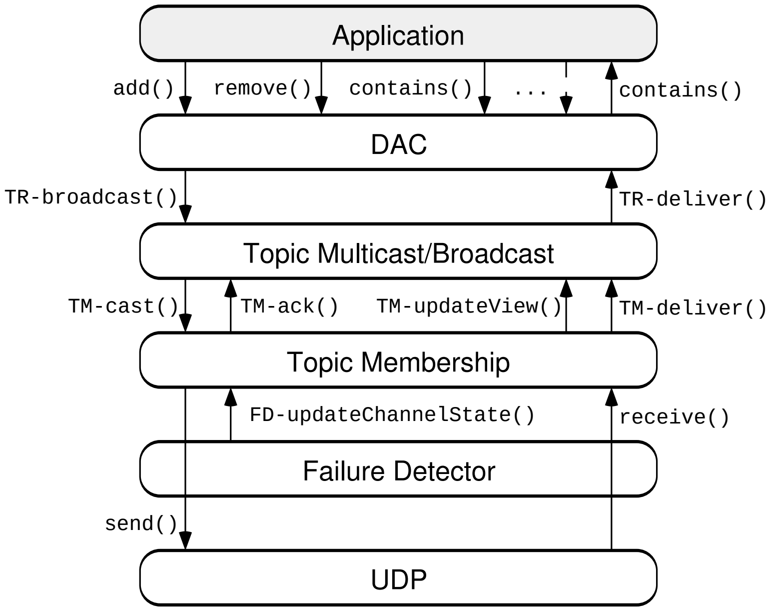


Figure 1. Layers

in a top-down order. They are presented in more detail in the following sections. As mentioned above, we focus on *topic-based* publish/subscribe in the context of this paper.

2.1. The application layer

Applications using the *DACE* publish/subscribe framework basically interact with a *DAC (Distributed Asynchronous Collection)*. The `add()` method for instance enables the addition of new objects to the collection, which comes to *publishing* new message objects. The interaction scheme shown in Figure 1 illustrates the *push* model where subscribers are called back (primitive `notify()`) upon incoming messages. However, *DAC*s offer a variety of possibilities of interacting with them, as we will see in Section 3.

2.2. The DAC layer

This layer is composed of the classes that implement the API of the *DAC* programming abstraction for publish/subscribe interaction. They are rather lightweight classes, which delegate general functionality to the underlying layer. Their tasks are similar to centralized container classes, i.e., they mainly take care of the local management of message objects.

Section 3 explains in more detail how a *DAC* represents a topic in the context of topic-based publish/subscribe.

The *DAC* applies a predefined threading model, by assigning notifications to threads. The class we use as an illustration in this paper is the `DAStrongSet` class, which guarantees *exactly-once* delivery semantics to a publisher. Published messages are passed to the underlying broadcast layer through the `TR-broadcast()` primitive, and messages are received through the `TR-deliver()` primitive.

2.3. The topic multicast/broadcast layer

This layer enables the multicast and broadcast of messages with different semantics to the subscribers of a topic. While the *Topic Broadcast* enables the broadcast of messages to *all* subscribers of a topic, the *Topic Multicast* is used in the context of content-based publish/subscribe [15]. As depicted earlier, a subscriber can delineate its individual requirements based on the properties of the messages. In such a scenario, a message must not be broadcast to all subscribers, but only to a subset, which proves the need for a multicast primitive. Section 6 gives an inside view of this layer focusing on broadcast issues.

Both broadcast and multicast come with *reliable*, *stubborn* [20] or *simple* (*best-effort*) semantics. This layer also takes care of broadcasting subscription information if a subscriber wants to join in or modify its subscription parameters. To send and receive messages, the subscriber uses the primitives `TM-cast()` and `TM-deliver()` respectively. The upper layer receives acknowledgements for successful message sends through `TM-ack()`.

2.4. The topic membership layer

The *Topic Membership* layer maintains a local view of the present and reachable subscribers for every given topic. The *Topic Membership* protocol is basically represented through the states of communication channels with other participants. This layer receives channel state updates either *locally* from the channel failure detector (FD) or *externally* from other processes, exactly like information about subscriptions and unsubscriptions. This layer indicates membership changes to the *Topic Multicast/Broadcast* layer with the primitive `TM-updateView()`, and sends and receives messages through the primitives `send()` and `receive()` of the *UDP* layer.

2.5. The failure detector layer

The *Channel Failure Detector* layer is used to administer a network topology and define the views of reachable subscribers. It is shared by several *DAC* instances hosted by the same process. Channel state changes as perceived by the failure detector are advertised to the *Topic Membership* layer through the `FD-updateChannelState()` primitive.

2.6. The UDP layer

Our entire publish/subscribe architecture is implemented on top of UDP. As conveyed by its name, UDP is a non reliable protocol, which offers the looseness required for the decoupled nature of publish/subscribe. Our Java implementation of *DACE* uses the standard Java classes for UDP sockets and datagrams (`java.net.DatagramPacket` and `java.net.DatagramSocket`), which are pretty close to the metal. These classes are wrapped into more powerful abstractions for communication channels (see Section 4).

3. DACE PROGRAMMING MODEL: A GENERAL SURVEY

This section gives a brief summary of our *DACE (Distributed Asynchronous Computing Environment)* framework for publish/subscribe interaction. We start by presenting the *Distributed Asynchronous Collection (DAC)* as programming abstraction, which enables the capture of the different styles of publish/subscribe (topic-based, content-based, type-based) without blurring their respective advantages. We then outline the interfaces related to topic-based publish/subscribe, and we show an overview of the corresponding classes.

3.1. Distributed Asynchronous Collections

Like the *group* abstraction which has been widely used as a basic model for replication [6], a *topic* enables the regrouping of several entities, which can thus be addressed atomically. For a publisher (in the case of group-based systems one could refer to an *invoker*) the set of subscribers appears as a single opaque entity, where subscribers remain anonymous to the application. Thanks to its *decoupled* nature, the publish/subscribe interaction model is the ideal way to express such *one-to-many* semantics at large scale.

3.1.1. DACs as object containers

Just like any collection, a *DAC* is an abstraction of a container object that represents a group of objects. It can be seen as a means to store, retrieve and manipulate objects that form a natural group. Unlike conventional collections or distributed collections described in [34] however, a *DAC* is not centralized on a single host, in order to guarantee its availability despite certain failures. In contrast, the distributed collections presented in [34] are centralized collections that can be remotely accessed through Java RMI.

3.1.2. The asynchronous flavor of DACs

Our notion of Distributed Asynchronous Collection represents more than just a distributed collection. In fact, a synchronous invocation of a distributed object can involve considerable latency, hardly comparable with that of a local interaction. Therefore we enforce an asynchronous interaction with our *DAC*s. By calling an operation of a *DAC*, one expresses an interest in *future notifications*. According to the terminology adopted in the *observer design pattern* [17], the *DAC* is the *subject* and its client is the *observer*. When querying a *DAC* for objects of a certain kind, the party interacting with the *DAC* expresses its interest in such objects. Therefore, when such an object is eventually "pushed" into the *DAC*, the interested party is asynchronously notified.

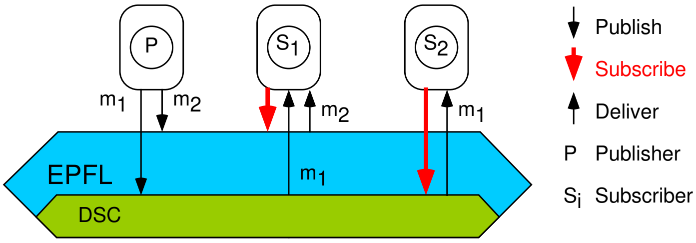


Figure 2. Topic-Based Publish/Subscribe with DACs

3.1.3. Topic-based publish/subscribe with DACs

Expressing ones interest in receiving information of a certain kind can be viewed as subscribing to information of that kind. By viewing *event notifications* as objects, a *DAC* can be seen as an entity representing related event notifications. Clearly, if a collection is a set of somehow related objects, a *DAC* can be seen as a set of related "events". When considering the classical topic-based approach to publish/subscribe, a *DAC* can be pictured as an extension of a conventional collection but also as a representation for a *topic*.

Such a topic is denoted by a name, like "EPFL". Topics can have specializations, or *subtopics*, and connecting to a topic requires the name in a URL-type format. Typically, "/EPFL/DSC" is a reference to the topic called "DSC" which is a subtopic of "EPFL". Subscribing to a topic can trigger subscriptions for the subtopics as well, as illustrated in Figure 2. Subscriber S_1 subscribes to topic "EPFL" and claims its interest in all subtopics. Hence S_1 does not only receive message m_2 but also message m_1 published for topic "/EPFL/DSC". In contrast, S_2 only subscribes to "/EPFL/DSC" and thus does not receive message m_2, which belongs to the *supertopic*.

Unlike other existing publish/subscribe systems (e.g., [22]), our approach frees the application programmer from the burden of marshalling and unmarshalling data into and from dedicated messages. In our context, a message can be basically any kind of object. In Java, this is expressed by allowing any object of class `java.lang.Object` to be passed as a message.[§]

[§]In order to be conveyable, a Java object should furthermore implement the `java.io.Serializable` interface [23], which contains no methods.

```
public interface DAC
  extends java.util.Collection

{
  public Object get();
  public boolean contains(Object m);
  public boolean add(Object m);
  ...
  public boolean contains(Notifiable n);
  ...
  public boolean containsAll(Notifiable n);
  ...
  public boolean remove(Notifiable n);
  ...
  public void clear(Notifiable n);
  ...
}
```

Figure 3. Interface DAC (Excerpt)

3.2. DAC interfaces

Figure 3 summarizes the main methods of the base `DAC` interface. More sophisticated interfaces like the `DASet` all derive from this interface, but are omitted for the sake of brevity. We roughly distinguish *synchronous* and *asynchronous* methods.

3.2.1. Synchronous methods

Since a *DAC* is in the first place a collection, the `DAC` interface inherits from the standard Java `java.util.Collection` interface. The inherited methods are adapted, and we denote them as *synchronous*. [15] gives more examples than shown here.

- `get()`. Similarly to a centralized collection, calling this method enables the retrieval of objects. This implements the *pull* model. Which element will be returned depends on the nature of the collection, as explained in [15].
- `contains(Object m)`. A *DAC* is first of all a representation of a collection of elements. This method enables the query of a collection for the presence of an object. Note that in the context of *topic-based* publish/subscribe, an object that is contained in a *DAC* belongs to (was published for) the topic represented by that *DAC*.
- `add(Object m)`. This method enables the addition of an object to the collection. The corresponding meaning for a *DAC* is straightforward: it allows to publish a message for the topic represented by that collection.

```
public interface Notifiable

{
  public void notify(Object m, String topicName);
}
```

Figure 4. Interface Notifiable

3.2.2. *Asynchronous methods*

We have added several *asynchronous* methods to express the decoupled nature of publish/subscribe interaction specific to *DAC*s. In these methods, asynchrony is expressed by an additional argument, denoting a *callback object* which implements the `Notifiable` interface given in Figure 4.

- `contains(Notifiable n)`. The effect, for instance, of invoking this method is not to check if the collection already contains an object revealing certain characteristics, but is to manifest an interest in any such object, that is eventually pushed into the collection. The interested party advertises its interest by providing a reference to an object implementing the `Notifiable` interface, through which it will be notified of events. There are different signatures for this method, among which certain enable for instance the specification of a filter for content-based subscribing.
- `containsAll(Notifiable n)`. This method offers the same signature(s) as the previous method. The difference is that a subscription is generated for all subtopics of the topic represented by this *DAC*. This conveys the situation of Figure 2.
- `remove(Notifiable n)`. By calling this method, a subscriber does not trigger the removal of an object already contained in the collection, but expresses its interest in being notified whenever an object matching its criteria is inserted in the collection, after which the object will be removed immediately. This expresses that a message is delivered to one single subscriber only . This is frequently called *one-for-all* or *one-of-n* [45] in contrast to *one-for-each* implemented by the two previous methods. Again there are several signatures for this method.
- `clear(Notifiable n)`. While the conventional argument-less `clear()` method enables the erasure of all elements from the collection, this asynchronous variant expresses the action of *unsubscribing*.

3.3. DAC classes

Our *DACE* framework consists of a variety of *DAC*s spanning different semantics and guarantees, since different applications have different requirements. These semantics can be seen as different *Qualities of Service (QoS)*. While some properties reflect in the interfaces, others concern the implementing classes (see Figure 5). Among those parameters is the *delivery semantics* of message objects "pushed" into the *DAC*. A related aspect is the possible occurrence of duplicates. Other parameters are more related to collections, like the order of storage, insertion or extraction of objects. We relate latter one to *pull* style interaction, and therefore omit the details in this paper.

3.3.1. Delivery semantics

When a producer publishes a message, it does not directly interact with subscribers. The details of the underlying multicast protocols are concealed, and might lead to different classes implementing the same interface. The `DASet` (Distributed Asynchronous Set) interface for instance is implemented by multiple classes. The first one does not offer more than plain unreliable delivery (`DAWeakSet`), whereas others guarantee reliability (e.g., `DAStrongSet`).

3.3.2. Duplicates

Just like it is possible to have duplicate elements in centralized collections, it is possible in *DAC*s that the same message is delivered more than once. The simple `DAWeakBag` class for instance does not prevent a notification from being delivered more than once, whereas the `DAWeakSet` class gives stronger guarantees by eliminating duplicate elements. This property is orthogonal to other characteristics of our *DAC*s.

3.3.3. Storage vs. delivery order

Collections are often characterized by the way they store their elements. *Sets* or *bags* do not rely on a deterministic order for their elements. Conversely, *sequences* can store their elements in an order given explicitly or implicitly based on properties of the elements. In *DAC*s however, the notion of *space* is somehow replaced by the notion of *time*. If some centralized collections reveal a deterministic storage order, a distributed asynchronous sequence may offer a deterministic ordering in terms of *order of delivery* to the subscribers. In the Java collection framework for instance, a *sorted set* is a sequence which is characterized by an ordering of the elements based on their properties. This can be seen as an implicit order. With our *DAC*s, an implicit order is a global delivery order on which the *DAC* itself decides. The `DASortedSet` class for instance presents a *total order* delivery. Inversely, a *FIFO* delivery order can be seen as an explicit order: it is given by the order in which events are notified to the *DAC* by a publisher.

3.3.4. Insertion order

In different centralized collections, the insertion order may have an impact on the storage order. A position can be given as an additional argument to an insertion into a *list* for instance. In an asynchronous collection however, the order of insertion corresponds to the order of publishing. It seems obvious that inserting an element at a specific position cannot translate to delivering a message at a certain moment in time relative to other messages: when inserting a message

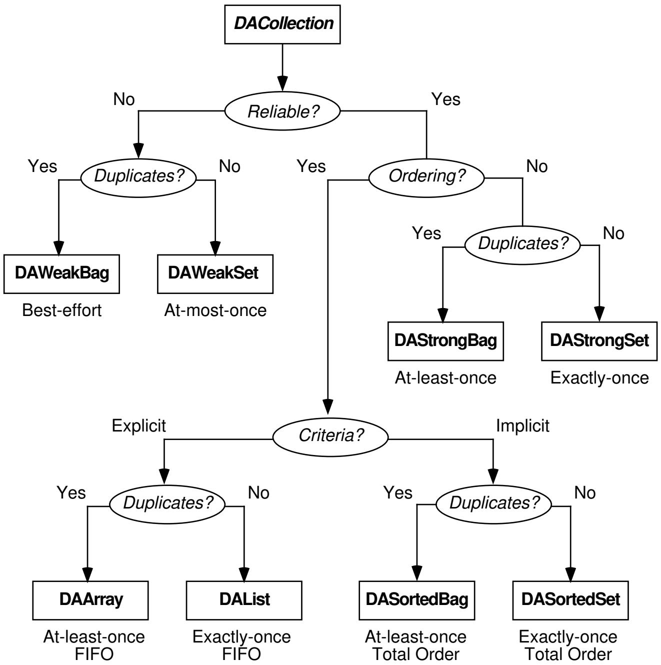


Figure 5. DACE Framework Classes

m at the beginning of a list, m would have to be sent before messages that have possibly already been delivered to subscribers. Therefore there is never any explicit argument for the order when "inserting" a new element into a *DAC*.

4. DACE SYSTEM MODEL

In order to describe the protocols used for the implementation of *DAC* classes and to prove their correctness, we first introduce the underlying system and failure model. We adopt a notation and a terminology similar the one introduced in [11]. We consider asynchronous message-passing distributed systems in which there is no bound on message delay, relative speed of processes, or the time necessary to execute a step.

The system is always considered with respect to a topic, since every topic is managed separately. The system consists of a finite set of processes or *topic participants*. A participant can act as *publisher*, *subscriber*, or as *both* for a given topic. It is then said to be a *participant* for that topic. A process can incorporate participants for several topics (it can *participate* in several topics). Our *communication* layer based on UDP implements (virtual) *channels* connecting pairs of participants, and furthermore offers the primitives `send()` and `receive()` (see Figure 1) for sending and receiving messages over them.¶ We use a discrete global clock whose range ticks T is the set of natural numbers. This notation is used to simplify presentation and not to introduce time synchrony since participants cannot access the global clock.

4.1. Participants

A topic involves a finite ordered set of n *topic participants* $\mathcal{T} = \{p_1, p_2, ..., p_n\}$. A participant p has a unique identifier denoted *p-id(p)*, and identifiers are ordered. We do not consider byzantine failures, i.e., participants do not behave maliciously. Participants can fail by crashing and may recover later. Formally: a *failure pattern* $F(t)$ of a topic is a function from T to $2^{\mathcal{T}}$, where $F(t)$ denotes the set of participants for that topic that do not run at time t. We say that participant p is *up at time t (in F)* if $p \notin F(t)$, and p is *down at time t (in F)* if $p \in F(t)$. We state that p *crashes at time t* if p is up at time *t-1* and p is down at time t. We can induce that p *recovers* at time $t \geq 1$ if p is down at time *t-1* and p is up at time t. We define *Correct(t)* as the set of participants that are *up at time t*.

4.2. History

At each clock tick, each participant p performs an event chosen from a set S. Set S includes at least the *null* event (denoted as ϵ) and the $send_p$ and $receive_p$ events, corresponding to the primitives `send()` and `receive()` depicted above. The global history of a run of a distributed algorithm is a function σ from $\mathcal{T} \times T$ to S. If a participant p executes an event $e \in S$ at time t, then $\sigma(p,t) = e$. If p executes no specific event at t, then $\sigma(p,t) = \epsilon$.

4.3. Channels

A participant p sends a message m to a participant q with the event $send_p(m,q)$, and receives a message m from q through the event $receive_p(m,q)$.

A communication channel between participant p and q is bidirectional but not FIFO (i.e., messages can be lost, duplicated, or unordered). If communication is possible from p to q at time t, then $p \rightarrow_t q$. A channel between p and q is said to be *open* at time t if the connection between p and q is *open on p and on q* at time t, and communication is possible in both directions. We denote this property $p \leftrightarrow_t q$. Intuitively, $p \leftrightarrow_t q \Leftrightarrow p \rightarrow_t q \wedge q \rightarrow_t p$. In any other case, a channel is *closed* at time t ($p \not\leftrightarrow_t q$). We assume that communication channels satisfy the following properties (which are formally proven in Appendix 2):

- *Eventual Symmetry.* If communication is possible from p to q, unless p or q crashes or they are partitioned, communication is eventually possible from q to p. Formally,

¶To make the model more comprehensible, two participants p and q each participating in topics x and y communicate through two distinct channels with each other; one for each topic. The implementation saves resources by using a single channel.

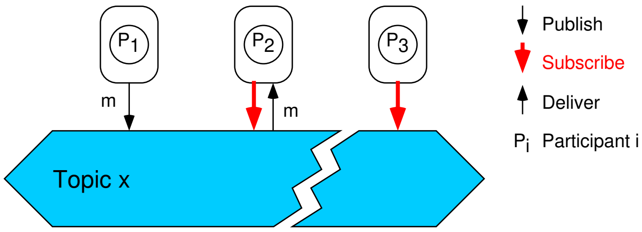


Figure 6. Partitions in a Topic

$$\exists t_0, \forall t \geq t_0 : p \rightarrow_t q \Rightarrow \exists t_1, \forall t' \geq t_1 : q \rightarrow_{t'} p$$

- *Fairness.* If $p \leftrightarrow_t q$, only one $send_p(m,q)$ from p is required for q to eventually receive m. This property can be guaranteed since our channels transparently resend messages as long as these have not been acknowledged by the recipient. Formally,

$$\forall t : p \rightarrow_t q \wedge \sigma(p,t) = send_p(m,q) \Rightarrow \exists t_0 \geq t : \ \sigma(q,t_0) = receive_q(m,p).$$

4.3.1. Channels and partitionings

Closed network links create communication failures which may partition the network. We assume that network partitions are only *temporary* and will be repaired *eventually*. We introduce the notion of *topic partitioning* as the effect of a network partitioning of the (sub)system composed of the participants of a topic. Figure 6 shows a simple scenario of a *partitioned topic*. Participants p_1 and p_2 can very well communicate, while p_3 is isolated from them. The sets $\{p_1, p_2\}$ and $\{p_3\}$ represent partitions, since they have no means of communicating with each other.

Communication links fail and recover more often than participants, and transitivity is not assured. As an example, we might have for a given t $p_1 \rightarrow_t p_2$ and $p_2 \rightarrow_t p_3$, but $p_1 \not\rightarrow_t p_3$.

4.3.2. Definitions

We define $Open_p(t)$ as the set of all *open* channels of p at time t, and $Closed_p(t)$ which denotes all closed channels of p at time t. Consequently, $Open_p(t) \cap Closed_p(t) = \emptyset$. Furthermore, we define:

Can Communicate With. Holds **true** at time t for p and q if there is a sequence of participants $p = p_0,...,p_{l+1} = q$ such that $\forall i \in [0,l]$, $p_i \leftrightarrow_t p_{i+1}$.[‖] We denote this relation by $p \rightsquigarrow_t q$. This

[‖]In fact, $\rightarrow_t$ is sufficient to guarantee this property.

relation indicates whether participant q can be *reached* by participant p at time t or not. If p cannot reach q, we will denote it as $p \not\rightsquigarrow_t q$.

Causal Message Chain. A causal message chain from p to q between t_0 and t_1 , noted $\mho_{p,q}(t_0, t_1)$, is a causal sequence of messages m_0,...,m_l and a sequence of participants $p = p_0$,...,$p_{l+1} = q$ such that: $\forall i \in [1, l]$ $\exists t_{i_0} < t_{i_1} : \sigma(p_i, t_{i_0}) = receive_{p_i}(m_{i-1}, p_{i-1})$, $\sigma(p_i, t_{i_1}) = send_{p_i}(m_i, p_{i+1})$ and $\exists t_p, t_q \in$ $[t_0, t_1]$: $\sigma(p, t_p) = send_p(m_0, p_1)$ and $\sigma(q, t_q) = receive_q(m_l, p_l)$.

4.4. Topic stability and partition

As described previously, communication channels can crash and recover. *Topic Stability* describes a stable state of the communication channels, while *Topic Partition* represents the partitioning of the system composed of the participants for a topic.

4.4.1. Topic stability and minimal topic stability

The state of the communication channels of a topic is *stable* from time t on, if the states of all communication channels between all participants of the topic do not change. In other words, all communication channels that are *open* at t_0 stay *open* and all communication channels that are *closed* at t_0 stay closed. Formally,

$$\forall t \geq t_0, \forall p \in \mathcal{T},\ Open_p(t) = Open_p(t_0) \wedge Closed_p(t) = Closed_p(t_0).$$

However, it is very unrealistic that a system remains stable forever. We derive from *topic stability*, a less restrictive property called *minimal topic stability* that assures stability for a certain period sufficient for a *causal message chain* to be established between every pair of participants in the system.** Formally,

$$\exists t_0, t_1, \forall t \in [t_0, t_1], \forall p \in \mathcal{T},\ Open_p(t) = Open_p(t_0) \wedge Closed_p(t) = Closed_p(t_0) \wedge \\ \forall (p, q)\ \exists\ \mho_{p,q}(t_0, t_1).$$

4.4.2. Topic partition

For a stable state of the communication channels, the relation $\rightsquigarrow$ defines an equivalence relation on the set of correct participants. The equivalence classes are called *partitions.* The partition of a participant p (the partition in which p is) at time t is denoted *partition(p,t)*.†† We can now define a partition pattern function P from $\mathcal{T} \times T$ to $2^{\mathcal{T}}$, where *P(p,t)* indicates at time t the set of participants that are not in the same partition as p. Formally, $P(p, t) = \{q \mid p \not\rightsquigarrow_t q\}$. If Figure 6 represents the situation for topic x at time t_0, then $P(P_3, t_0) = \{P_1, P_2\}$.

**To simplify, we could also require a message to be exchanged between every pair of participants. However, the total number of messages sent would be greater or equal than with the *causal message chain* approach.

††If $p \in F(t)$ then *partition(p,t)* $= \emptyset$.

5. TOPIC MEMBERSHIP

We have designed the protocols underlying our implementation of *DACE* to manage partitioning *as well as* crashes. The marriage of large scale, high throughput and fault tolerance has led us to consider weak consistency protocols. This section presents *Topic Membership*, which can be viewed as a lightweight membership protocol for the participants of a topic. First, we introduce our notions of *topic view* and *stable topic view*. We then describe our *Channel Failure Detector* and its properties. Finally, we formally define our notion of *Topic Membership*.

5.1. Views

Topic Membership is a weakly consistent membership notion which is different from the traditional notion of *group membership* [39]. Approaches like *virtual synchrony* [7] offer strongly consistent views, but do not scale well. Our notion of *view* is less restrictive, i.e., there is no explicit agreement on views. We distinguish between two kinds of views: the *topic view* and the *stable topic view*. In fact, the system can be viewed as a sequence of alternations of *stable* and *unstable* (*stabilization*) phases. Latter ones begin with the occurrence of failures, and may result in differences in local views. Eventually, the views of the participants inside a partition converge to a *stable topic view*.

5.1.1. Topic view

The *topic view* corresponds to the local participant view for a topic and reflects the participant's perception of reachable and present participants. These views resemble the views defined by [4] by being concurrent. A *topic view* is bound to a single topic, and a process which participates in different topics maintains separate views for each topic. Note that subtopics are handled like independent topics, which implies that a *topic view* is required for each (sub)topic.

5.1.2. Stable topic view

Once the system is in a stable phase and views of the participants inside the partition have converged, the participants are said to have reached a *stable topic view*. To achieve a *stable topic view*, the *system* must undergo *minimal topic stability*.

A *stable topic view stv* represents a set of participants. $stview(p, t)$ represents the last *stable topic view* that was reached by p before time t. If stv_j *succeeds* stv_i at p, then $stv_i \prec_p stv_j$. Formally,

$$\exists t_0 \leq t_1 \leq t_2, \forall t \in [t_0, t_2] : Open_p(t) = Open_p(t_0) \wedge Closed_p(t) = Closed_p(t_0) \wedge \\ \forall(p,q)\ \exists\ \mho_{p,q}(t_0, t_1) \Leftrightarrow \exists\ stv, \forall t' \in [t_1, t_2] : stview(p, t') = stv.$$

5.2. Topic channel failure detector

Each participant p has access to a local failure detector module which outputs hints about the closed channels of p with other participants. The topic channel failure detector history CH is a function from $\mathcal{T} \times T$ to $2^{\mathcal{T}}$ that outputs the closed channels of the participant. Formally,

$$q \in CH(p,t) \Leftrightarrow p \not\to_t q; \quad q \notin CH(p,t) \Leftrightarrow p \to_t q.$$

We assume that the topic channel failure detector is perfect with respect to our (virtual) channels. A channel loss due to a failure in the network is always detected eventually. If the failure affected the existing connection, but the network still offers a correct physical path between the participants, the channel will be re-established. The same action takes place in the case of false suspicion. During such *glitches*, the system is considered being in an *unstable* phase.

5.3. Topic membership specification

As explained previously, a *topic view* represents a participant's view of all participants of a topic at any moment. We have shown that when the system is stable long enough to satisfy *minimal stability*, the views of all participants of a topic inside a partition (or the entire system) are identical. The view becomes *stable*, and is hence called *stable topic view*. In contrast to [4], which specifies a membership based on properties of local views, we specify our *Topic Membership* by properties of *stable topic views*, and do not consider inconsistent views, since these correspond to *unstable* phases.

(TM1) *Stable Topic View Agreement.* If participant p reaches *stable topic view* stv_1 and its immediate successor stv_2, both containing q, then p reaches stv_2 after q reached stv_1. Formally,

$$\forall p, q, stv_1, stv_2 : stv_1 \prec_p stv_2 \wedge p, q \in stv_1 \cap stv_2 \wedge \forall t \in [t_0, t_1], sview(p,t) = svt_1 \wedge \\ \forall t' \in [t_2, t_3], sview(q,t') = svt_1 \Rightarrow t_2 < t_1.$$

(TM2) *Stable Topic View Accuracy.* If $p \rightsquigarrow q$ holds forever, then eventually when the system reaches *stable topic view stv*, p and q eventually have the same *view*. Formally,

$$\exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \Rightarrow \exists t_1, \forall t' \geq t_1 : sview(p,t') = sview(q,t').$$

(TM3) *Stable Topic View Completeness.* If all processes q in some partition Ω hold $p \not\rightsquigarrow q$ forever, then eventually when the system reaches *stable topic view*, p will not have any processes $q \in \Omega$ in its *stable topic view*. Formally,

$$\exists t_0, \forall q \in \Omega, \forall p \notin \Omega, \forall t \geq t_0 : p \not\rightsquigarrow_t q \Rightarrow \exists t_1, \forall t' \geq t_1 : sview(p,t') \cap \Omega = \emptyset.$$

(TM4) *Stable Topic View Integrity.* Every participant p that reaches a *stable topic view* is included in that *stable topic view*. Formally,

$$\forall p, t : p \in sview(p,t).$$

6. TOPIC RELIABLE BROADCAST

This section sketches the properties of our *Topic Reliable Broadcast* protocol, which is used to efficiently and reliably multicast messages despite partitionings. *Topic Reliable Broadcast*, hereafter called *TR Broadcast*, is based on *Topic Membership*, and enables the broadcasting of messages to all subscribers of a topic. The realization of the `DAStrongBag` class, shown in Figure 5, is based on this protocol. The simplified algorithm is given in Appendix 1.

6.1. Specification of TR broadcast

We recall the properties of reliable broadcast (in the sense of [11]). It guarantees that (a) all correct processes deliver the same set of messages, (b) all messages broadcast by correct processes are delivered and (c) no spurious messages are ever delivered. These properties can be transposed to partitioning and topics. Formally, our notion of *TR Broadcast* (Topic Reliable Broadcast) is based on the two primitives *TR-broadcast* and *TR-deliver*, which satisfy the following properties:

(a) Validity. If a correct publisher p *TR-broadcasts* a message m, then unless p crashes, a correct subscriber eventually *TR-delivers* m.

(b) Agreement. If a correct subscriber s *TR-delivers* a message m, then all correct subscribers eventually *TR-deliver* m.

(c) Uniform Integrity. For any message m and any subscriber s that *TR-delivers* m, s *TR-delivers* m at most once and only if m was previously *TR-broadcast* by publisher(m).

6.2. General concepts

The overall goal of *TR Broadcast* is to ensure that a message broadcast by a publisher reaches *all* subscribers of the topic. For that purpose, we require the knowledge of *identifiers* of the received messages of each participant. We introduce here the general concepts of our algorithm.

6.2.1. Messages

Each application message m has a unique identifier, denoted *m-id(m)*. Messages are composed of two fields. The `data` field carries application messages. The `control` field carries updates of the states of the communications channels (see Section 7 for more details) as well as identifiers of received messages for every participant. These acknowledgements are used especially for garbage collection. By piggybacking them with other messages we reduce the overall network traffic.

6.2.2. First participant

When a participant p receives a message m, it tries to determine for every neighbour participant q with which it has a channel if it is the participant with the lowest identifier that has received m and has a channel with q. If these conditions are fulfilled, p will forward m to q. This reduces the amount of redundant message transfers, without violating the *Agreement* property. For a

Prepared using cpeauth.cls

participant q and a message m, there is only one *first participant* p in the whole system which will send m to q.[‡‡]

6.2.3. *Check & forward*

In the case of remerging partitions, participants who were in different partitions must exchange the messages they have delivered in the meantime. Therefore, upon changes in the state of communication channels, participant p checks for every participant q which messages p has received and q has not acknowledged to p. Process p then forwards every such m to q if p is *first participant* of q with respect to m. This way, we ensure that all messages are received eventually despite *unstable* phases.

6.2.4. *Subscriptions and unsubscriptions*

When a process wants to subscribe to a topic, it must know at least one participant p, which will reliably broadcast a subscription request vicariously for the new participant q. Participant q will receive all messages that p receives after q's subscription request. When unsubscribing, a participant reliably broadcasts an unsubscription request, which guarantees that every participant will receive it. Note, that neither the subscription nor the unsubscription of a participant requires any agreement protocol.

7. IMPLEMENTATION

This section depicts some implementation issues of *DACE* and illustrates the performance of *TR Broadcast*. This gives an idea of the overall efficiency of our protocols.

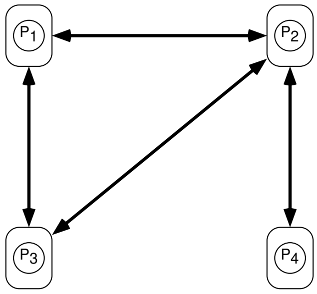
7.1. Topic network knowledge

We call *topic network knowledge* the information that a participant p has about the states of all channels between participants of the topic. To learn about the states of all channels connecting participants of the topic, participants must exchange their information.

7.1.1. *Topic channel state*

The information p has about all the channels between participants of the topic are stored in a $n \times n$ matrix called *channelState*$_p$. The value of $channelState_p[q, r]$ represents the state of the channel between q and r ($q \to r$) as assumed by p. The matrix $channelState_p$ is divided in n *channelState vectors*, each corresponding to a line of the *channelState*$_p$ matrix. *channelState*$_p$*(q)* is the *q-th channelState vector* of participant p. It represents p's view of the

[‡‡]In fact, there is exactly one in *stable* phases. In *unstable* phases, there might be more than one.



(a) Link Status

	p1	p2	p3	p4
p1		○	○	×
p2	○		○	○
p3	○	○		×
p4	×	○	×	

(b) *channelState*

Figure 7. *channelState* derived from the Link Status

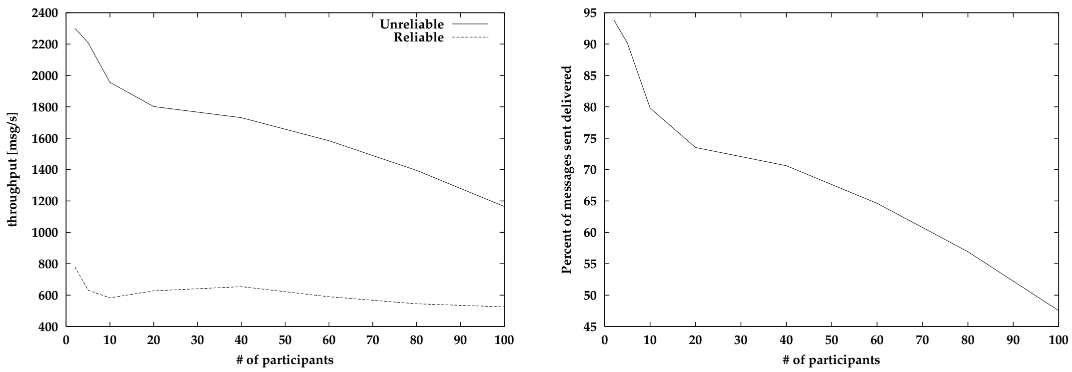
channels q has all with other participants. A logical timestamp $ts_p(q)$ is associated with each $channelState_p(q)$. Figure 7 shows a typical *channelState* matrix in a stable system where all participants share the same *channelState*; $\times$ means that the link is *closed* or does not exist and $\bigcirc$ that the link is *open*.

7.1.2. Propagation of knowledge

When participant p sends a message m to q, p checks if *channelState*$_p$ has changed since the last message sent to q. This happens whether p actually published the message itself or only forwards it. Message m piggybacks the updated *channelState*$_p$ (with the associated updated timestamps). When q receives m from p, q compares all received timestamps and replaces all *channelState vectors* that are older. In the absence of application messages, each participant p periodically sends its own *channelState*$_p$ matrix to a randomly picked neighbour (gossip). The receiver q updates its own *channelState*$_q$ matrix, and sends its more up-to-date values to p. This keeps the *channelStates* from diverging when no messages are published for a certain time. When participant p receives a message from a new participant, p increases the size of *channelState*$_p$.

7.2. Performance

We give here performance measurements of our prototype which were made on two LANs interconnected by Fast Ethernet (100MB/s) on normal working days. The first LAN consisted of 60 SPARCstation 20 (model 502: 2 SuperSPARC CPU, 64Mb RAM, 1Gb Harddisk) machines, and the second one of 60 UltraSUN 10 (256Mb RAM, 9 Gb Harddisk) machines. All stations were running Solaris 2.6, and *DACE* was running on Solaris JVM (JDK 1.2.1., native threads, JIT). The message objects were of a size of 1Kb in serialized form. Figure 8(a)



(a) Unreliable vs Reliable Topic Broadcast

(b) Unreliable Topic Broadcast Loss Rate

Figure 8. Performance of Unreliable and Reliable Topic Broadcast

summarizes the results of the throughput measurements and compares *TR Broadcast* with an unreliable broadcast in a topic. Figure 8(b) shows the percentage of sent messages that are delivered by the unreliable broadcast algorithm. The complete results can be found in Appendix 3.

As conveyed by the measurement results, the performance of *TR Broadcast* remains stable over an increasing number of participants. After 100 participants, the performance varies very little. On the other hand, the performance of the unreliable broadcast is less stable. It is limited by the overall performance of the network, which can be seen by the quickly decreasing throughput. When the number of participants exceeds 100, the two curves converge, since the *TR Broadcast* protocol reaches the limits of the network earlier.

8. RELATED WORK

In the past few years, the need for effective large scale multicast interaction schemes and protocols have been widely recognized and much effort has therefore been invested in this domain. A multitude of approaches have emerged from academic as well as industrial researches. We present here the main characteristics of these approaches and we compare them with our *Distributed Asynchronous Computing Environment.*

8.1. Publish/subscribe messaging systems

In order to integrate the publish/subscribe communication style into existing middleware standards, specifications have been conceived by both the Object Management Group [36] and Sun [22, 1, 8]. The OMG's CORBA service for publish/subscribe-oriented communication, called the *CORBA Event Service*, is based on the notion of *event channels*. These channels are denoted by names, and basically incorporate topics. In all implementations we know about, channels are centralized components and therefore manifest a strong sensitivity to any component failure, which makes them unsuitable for critical applications. The *Java Messaging Service* [22] is a specification from Sun. Its goal is to offer a unified Java API around common publish/subscribe engines. Certain existing services implement the JMS, but to our knowledge no publish/subscribe system has been implemented with the mere goal to support the JMS API directly. Its generic nature, required in order to match a maximum number of existing systems, appears to be rather cumbersome. Other specifications from Sun are more aimed at particular environments, like the *Java Distributed Event Specification* [1] in the context of *Jini* and the *InfoBus 1.2 Specification* [8] describing an information bus for dynamic data exchange between *JavaBeans*.

Established industrial strength solutions, like *TIB/Rendezvous* [45] or *Smartsockets* [9] tolerate crash failures by applying entity redundancy, and Smartsockets even take network failures into account. Nevertheless, even though such solutions might offer fault tolerance, they provide a rather complicated programming model.

Systems like *Siena* [10], *Elvin* [43] or *Gryphon* [2] provide a flexible programming model with content-based capabilities. These solutions however focus on the effective dissemination of information, without sufficiently addressing fault tolerance.

JGL [34] was designed to provide a more advanced series of collections, since the Java environment by default only offers limited support for data collections and algorithms. JGL extends the basic Java collections with more refined types. The notion of distributed collection in JGL though describes a centralized collection object, accessible through Java RMI. This is especially prone to failures, while *DAC*s are especially designed for fault tolerance.

Clearly, none of these solutions provides a generic approach to publish/subscribe interaction. *DACE* introduces an easy to use high-level programming abstraction which enables the grouping of different styles without blurring their advantages. At the same time, our framework blends these different publish/subscribe variants with a multitude of QoS among which certain offer strong reliability guarantees without penalizing efficiency. This allows for instance to easily realize a JMS-compliant service on top of *DAC*s. Inversely, *DAC*s could, to some extent, also be build on top of a JMS implementation. In particular, the *selector* concept in JMS could be used to express the content-based features of *DAC*s [16], and the generally weak specification of JMS would enable the translation of QoS and subtopics in *DAC*s (the JMS `Destination` is represented by an empty interface).

8.2. Network partition models

Several partition models have emerged, advocated by different types of applications. The requirement for strong consistency, for instance in database or file system applications [18], has driven an approach where services have to be suspended completely in all but one partition that contains a majority of processes. This is known as following the *primary-partition* model, adopted for instance by Isis [6], Amoeba [27], [33], or [41]. The overhead introduced by strong consistency is not well adapted to large scale, and such systems might block as long as the majority condition is not satisfied. Furthermore, members of minority partitions are forced to quit applications.

Applications relying on mobile units or wireless links as well as application at large scale are forced to consider more than one partition; they often follow the *minority-partition* model (or *partitionable model*). Partitions occur when units are deliberately (or unintentionally) unplugged from the network. Applications such as the *Coda* [26] or the *Ficus* file systems [38], and *Rover* [25] rely on that model, and furthermore apply to large scale. In contrast to the previous model, this class of applications must be able to make progress without blocking even under numerous partitions.

[4] introduces *partition-aware* applications. The system provides the necessary hooks such that the application itself decides which of its services will be available in each partition and at what QoS levels. *Total order* delivery is possible in concurrent partitions, and the states of the objects in different partitions are merged when the partitions remerge [32]. This however forces the applications to offer such merging facilities, which constitutes a non-trivial issue.

Recently, many attempts have been made to formalize a specification of a partitionable group membership in asynchronous systems. [4] presents a formal specification for partitionable group membership and its algorithms. Another known attempt is [13]. Systems such as Horus [46], Transis [14], or Totem [31] manage *minority-partitions.* They handle concurrent views in different partitions. The membership incorporated by those approaches however introduces important overheads in order to guarantee a consistency which is too strong for our case.

The model underlying our environment differs from the above proposals in many aspects. First, our model is based on an unreliable datagram transport. Second, the *Topic Membership* model is less restrictive in the sense that no consensus is required and it does not enforce view changes. Third, most of these system models (*primary-partition* and *minority-partitions*) are applied to local area networks and do not scale well to wide area networks. The *partition-aware* model is aimed at large scale, but requires considerable support from the application for the state merging of participants who were partitioned. It is aimed at more specific applications, like replication, while *DACE* is a generic messaging system, in which we do not consider "states" of participants.

9. SUMMARY AND CONCLUSIONS

Current research in the context of distributed computing encompasses two major and often separated trends: *distributed algorithms* and *distributed programming models.* The first trend is strongly guided by the development of sophisticated protocols for a multitude of semantics or QoS based on a variety of different systems and failure models. As an example, the need for multicast primitives tailored to large scale environments, like the Internet, has been recognized and has led to a variety of protocols with diverse semantics. Such protocols are often developed without considering the look-and-feel in which they will be enclosed, i.e., the actual programming model. Applications that want to benefit from such facilities are hence bound to rather primitive and unwieldy services which are close to the metal.

On the other hand, the practice of distributed computing models is largely driven by the desire of handling distribution as an *implementation issue*, i.e., all aspects related to distribution are hidden behind traditional centralized constructs. This has led to a variety of so-called "object-oriented middleware" solutions, which promote objects as "autonomous entities communicating via message passing". One fundamental idea behind this is the illusion to be able to reuse, in a distributed context, a centralized program that was designed and implemented without distribution in mind.

As argued in [47, 29, 21], distribution transparency is a myth that is both misleading and dangerous. Distributed interactions are inherently unreliable and often introduce significant latency that is hardly comparable to that of local interactions. The possibility of partial failures can fundamentally change the semantics of an invocation. High availability and masking of partial failures involves distributed protocols that are expensive and hard to implement in the presence of network partitions. Conventional protocols might conform well to local area networks, but scale poorly. Another important mismatch lies in the missing support for *one-to-many* invocations in middleware based on client/server-like interactions.

We have been considering an approach that bridges the gap between the two trends: (1) distributed protocols and (2) distributed programming models. In our approach the programmer is aware of distribution but the ugly and complicated aspects of distribution are encapsulated inside a specific abstraction with a well-defined interface. The *Distributed Asynchronous Collection* [15] is such an abstraction. It is a simple extension of the well-known collection abstraction. *DAC*s add an asynchronous and distributed flavor to traditional collections [5], and enable the expression of various forms of publish/subscribe interaction. The *Distributed Asynchronous Computing Environment* we present in this paper is a framework for large scale event dissemination based on *DAC*s, and can be viewed as a middleware solution for publish/subscribe interaction.

We define an adequate underlying system and failure model for the implementation of our *DAC*s. This allows us to seamlessly weave programming models and underlying protocols, instead of just gluing them together. This paper exemplifies this by presenting the realization of a *DAC* class for *topic-based* publish/subscribe from bottom to top, i.e., from the system model all the way up to the resulting programming abstraction and its interface. The `DAStrongSet` class used as an illustration, guarantees reliable event delivery to all subscribers of a topic, and demonstrates our underlying partitioning model *made-to-measure* for loosely coupled interaction at large scale. Our model is less restrictive than *majority-partition*, *minority-partition* or *partition-aware*, but nevertheless guarantees a useful reliability.

The *Topic Membership* protocol we present in this paper is a lightweight membership protocol for topics. It was guided by our programming model, namely *DAC*s. It handles network partitions in asynchronous distributed systems. It makes no assumption on the network used to transport messages, except that it must guarantee the absence of byzantine failures, i.e., processes do not behave maliciously. The *Topic Reliable Broadcast* protocol provides a reliable broadcast for topic-based publish/subscribe, under the assumption that partitions will eventually remerge. A subscriber will eventually receive the message even if the publisher was partitioned away temporarily. The number of messages sent is minimized in stable phases by using the *first participant* function. We have also made broad use of an augmented version of that function for our multicast protocol [3].

REFERENCES

1. K. Arnold and B. O'Sullivan and R.W. Scheifler and J. Waldo and J. Wollrath. The Jini Specification. Addison Wesley, 1999.
2. G. Banavar and T. Chandra and B. Muhkerjes and J. Nagarajarao and R.E. Strom and D.C. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS-99).
3. R. Boichat and L. Duchien. Reliable Broadcast and Multicast tolerant to partitions. In Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and System, Boston, USA, 1999.
4. Ö. Babaoğlu and R. Davoli and A. Montresor and R. Segala. System Support for Partition-Aware Network Applications. Technical Report UBLCS 97-08, Department of Computer Science, University of Bologna, 1997.
5. J.-P. Briot and R. Guerraoui and K.-P. Löhr. Concurrency and Distribution in Object-Oriented Programming. In ACM Computing Surveys, September 1998.
6. K.P. Birman. The Process Group Approach to Reliable Distributed Computing. In Communications of the ACM, 36(12):36-53, December 1993.
7. K.P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In Proceedings of the 11th ACM Symposium on Operating Systems Principles, pages 123-138, 1987.
8. M. Colan. InfoBus 1.2 Specifications. Sun Microsystems Inc., February 1999.
9. Talarian Corporation. Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper). http://www.talarian.com/, 1999.
10. A. Carzaniga and D.S. Rosenblum and A.L. Wolf. Design of a Scalable Event Notification Service: Interface and Architecture. Technical Report, Department of Computer Science, University of Colorado, 1998, http://www.cs.colorado.edu/ carzanig/papers/.
11. T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2):225-267, 1996.
12. E.W. Dijkstra. Self-stabilizing Systems in spite of Distributed Control. Communications of the ACM, 17:643-644, 1974.
13. D. Dolev and D. Malki and R. Strong. A framework for partitionable membership service. In Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-96), Philadelphia, PA, USA, 1996.
14. D. Dolev and D. Malki. The Transis Approach to high-availability cluster communication. Communications of the ACM, 39(4), april 1996.
15. P.T. Eugster and R. Guerraoui and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP-00), pages 252-276, june 2000.
16. P.T. Eugster and R. Guerraoui Content-Based Publish/Subscribe with Structural Reflection. To appear in Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS-01), january 2001.
17. E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
18. H. Garcia-Molina and S.B. Davidson and S. Skeen. Consistency in partitioned networks. ACM Computing surveys: 17(3):341-370, 1985.
19. R. Golding. . A weak-consistency architecture for distributed information services. Computing Systems, 5(4):179-205, 1992.
20. R. Guerraoui and R. Oliveira and A. Schiper. Stubborn Communication Channels. Technical Report, Ecole Polytechnique Fédérale de Lausanne, 1997.
21. R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. In Informatik, 2, April 1999.
22. M. Happner and R. Burridge and R. Sharma. Java Message Service. Sun Microsystems Inc., October 1998.

23. Sun Microsystems Inc. The Java Platform 1.2 API Specification. http://java.sun.com/products/jdk/1.2/, 1999.
24. Sun Microsystems Inc. The Java Collections Framework. http://java.sun.com/products/jdk/1.2/, 1999.
25. A.D. Joseph and A.F. deLespinasse and J.A. Tauber and D.K. Gifford and M.F. Kaashoek. Rover: A toolkit for mobile information access. In Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP-95), pages 156-171, 1995.
26. J.J. Kitsler and M. Satyanarayanan. Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, 10(1):3-25, 1992.
27. M.F. Kaashoek and A. Tanenbaum. Communication in the Amoeba distributed Operating system. In Proceedings of 11th IEEE International Conference on Distributed Computing Systems (ICDCS-91), pages 222-230, 1991.
28. C. Labovitz and A. Ahuja and F. Jahanian. Experimental Study of Internet Stability and Wide-Area Backbone Failures. Technical Report CSE-TR 382-98, Department of Eletrical Engineering and Computer Science, University of Michigan, 1998.
29. D. Lea Design for open systems in Java. In Second International Conference on Coordination Models and Languages, 1997. http://gee.cs.oswego.edu/dl/coord/.
30. Microsoft Co. DCOM Technical Overview (White Paper), 1999.
31. L.E. Moser and P.M. Meillar-Smith and D.A. Agarwal and R.K. Budhia and C.A. Lingley-Papadopoulos. Totem : A Fault-Tolerant Multicast Group Communication System. Communications of the ACM, 39(4), 1996
32. A. Montresor. Jgroup. Phd Thesis, University of Bologna, 2000.
33. S. Mishra and L. Peterson and R. Schlichting. A membership protocol based on partial order. In Proceedings of the International Workshop on Parallel and distributed Algorithms, pages 137-145, 1991.
34. ObjectSpace. JGL - Generic Collection Library. http://www.objectspace.com/products/jgl/, 1999.
35. OMG. The Common Object Request Broker: Architecture and Specification. February 1998.
36. OMG. CORBAservices: Common Object Services Specification. December 1998.
37. B. Oki and M. Pfluegl and A. Siegel and D. Skeen The Information Bus - An Architecture for Extensible Distributed Systems. In Proceedings of the 14th ACM Symposium on Operating System Principles, pages 58-68, December 1993
38. G. Popek and R. Guy and T. Page and J. Heidemann. Replication in Ficus Distributed file systems. In the Workshop on Management of Replicated Data, pages 5-10, 1990.
39. D. Powell. Group Communications. In Communications of the ACM, 39:4, pages 50-97, April 1996.
40. K. Petersen and M.J. Spreitzer and D.B. Terry and M.M. Theimer and A.J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-97), pages 288-301, 1997.
41. A.M. Ricciardi and K.P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-91), pages 341-352, 1991.
42. D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, September 1997.
43. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG-97), 1997.
44. Sun Microsystems Inc. Java Remote Method Invocation - Distributed Computing for Java (White Paper). http://java.sun.com/marketing/collateral/javarmi.html/, 1999.
45. TIBCO Inc. TIB/Rendezvous White Paper. http://www.rv.tibco.com/whitepaper.html, 1999.
46. R. van Renesse and K.P. Birman and S. Maffeis. Horus : A Flexible group communication system. Communication of the ACM, 39(4), 1996.
47. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. A Note on Distributed Computing. Sun Microsystems Inc., November 1994.

10. Appendix 1: Topic reliable broadcast protocol

This section presents the *TR Broadcast* algorithm without the parts related to gossiping of network knowledge and acknowledgements (*anti-entropy*).

```
this-p: this participant
channelState(i,j): channel exists between i and j as seen by this-p (true or false)
channelState(i): i-th row of channelState; has a timestamp tsp(channelState(i)) associated
messagesReceived: set of messages received by this-p
idMessagesReceived(q): set of ids of messages received by participant q as seen by this-p
maxGarbagedId(q): highest id of messages received by all and published by q as seen by this-p
awaitedIdMessages(q): set of ids of messages that have not yet been acknowledged by all

function firstParticipant(m,q)        {return true if this-p is the firstParticipant to send m to q}
  neighbours = {r | channelState(r,q) is true}        {ordered with increasing p-id()'s}
  for all p-id(r) | r ∈ neighbours up to p-id(this-p) - 1 do
    if m-id(m) ∈ idMessagesReceived(r) then
      return false
  return true

procedure updateIdMessages(idMessagesReceived_q, maxGarbageId_q, awaitedIdMessages_q, q)
  for all participant r ≠ this-p do        {update ids}
    idMessagesReceived(r) ⟵ idMessagesReceived(r) ∪ idMessagesReceived_q(r)
  for all id ∈ maxGarbageId_q \ awaitedIdMessages_q do
    idMessagesReceived(q) ⟵ idMessagesReceived(q) ∪ id

procedure check&forward()        {upon a change in channelState}
  neighbours = {r | channelState(r,q) is true}
  for all r ∈ neighbours do
    for all m ∈ messagesReceived and m-id(m) ∉ idMessagesReceived(r) do
      if firstParticipant(m,r) then
        TM-cast(m, idMessagesReceived, r)

To execute TR-broadcast(m):
  messagesReceived ⟵ messagesReceived ∪ m
  idMessagesReceived(this-p) ⟵ idMessagesReceived(this-p) ∪ m-id(m)
  neighbours = {r | channelState(r,q) is true}
  for all r ∈ neighbours do
    TM-cast(m, idMessagesReceived, maxGarbageId, awaitedIdMessages, r)
  TR-deliver(m)        {delivers m to itself}

TR-deliver(-) occurs as follows:
  when TM-deliver(m, idMessagesReceived_q, maxGarbageId_q, awaitedMessages_q, q)
  if m-id(m) > maxGarbagedId(q) and m-id(m) ∉ awaitedIdMessages(q) then
    updateIdMessages(idMessagesReceived_q, maxGarbageId_q, awaitedIdMessages_q, q)
    if m ∉ messagesReceived then
      messagesReceived ⟵ messagesReceived ∪ m
      idMessagesReceived(q) ⟵ idMessagesReceived(q) ∪ m-id(m)
      TR-deliver(m)        {delivers m}
    check&forward()
```

```
To execute TM-cast(m, idMessagesReceived, maxGarbageId, awaitedMessages, q):
  for all participant r do
    if [channelState(r) changed since the last message sent to q] then
      tsp(channelState(r)) = tsp(channelState(r))+1                {update tsp}
      channelState_q ⟵ channelState_q ∪ channelState(r)            {update channelState}
  send(m, idMessagesReceived, maxGarbageId, awaitedMessages, channelState_q) to q

TM-deliver(-) occurs as follows:
  when receive(m,idMessagesReceived_q,maxGarbageId_q,awaitedMessages_q,channelState_q) from q
  for all [participant r | ∃ channelState_q(r)] do
    if tsp(channelState_q(r)) < tsp(channelState(r)) then
      channelState(r) = channelState_q(r)
      tsp(channelState(r)) = tsp(channelState_q(r))
  TM-deliver(idMessagesReceived_q, maxGarbageId_q, awaitedMessages_q, q)

when messagesReceived ≠ ∅
  for all m ∈ messagesReceived do
    if [∀ participant r | m-id(m) ∈ idMessagesReceived(r)] then
      messagesReceived ⟵ messagesReceived \ m
      for all participant r do
        idMessagesReceived(r) ⟵ idMessagesReceived(r) \ m-id(m)
      if maxGarbagedId(m) > m-id(m) then
        awaitedIdMessages(m) ⟵ awaitedIdMessages(m) \ m-id(m)
      else
        maxGarbagedId(m) = m-id(m)
        for all m-id(m) > j > maxGarbagedId(m) do
          awaitedIdMessages(m) ⟵ awaitedIdMessages(m) ∪ j
```

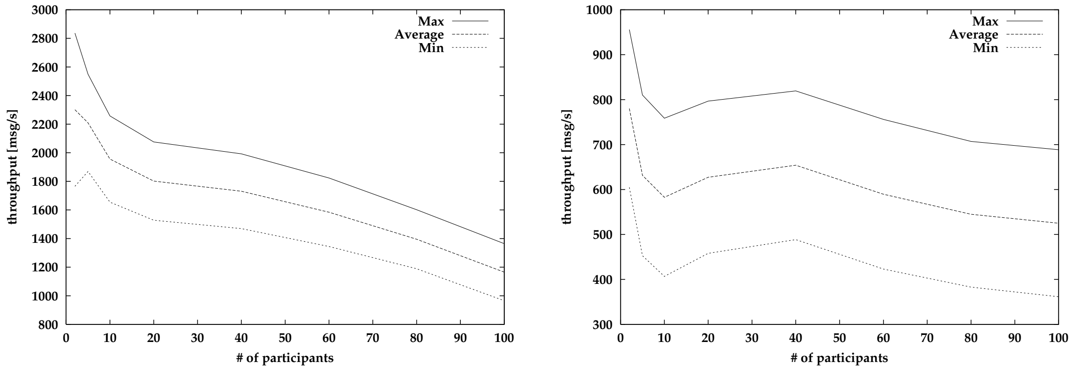
11. Appendix 2: Topic reliable broadcast properties

This appendix sketches the properties of our *TR Broadcast* algorithm given in Section 6. For this, we suppose that the system has reached an agreement on *channelState*.

Validity is implicit since at line 31 p delivers it directly by appending m to *messagesReceived*. That way, when p *TR-broadcasts* m, p will automatically *TR-deliver* m. Even if p does not incorporate a subscriber, the message will still be buffered.

Agreement is fulfilled if one participant p eventually *TR-delivers* m, then every participant q of the topic delivers m. Partitions remerge eventually, and therefore there will be a time t at which $p \leadsto_t q$. A message m is only garbage collected when all neighbours have acknowledged it (line 56), and the algorithm will forward missing messages to lagging processes in task $check\&forward_p$. Therefore, every participant (and thus every subscriber) will eventually *TR-deliver* m.

Uniform Integrity is ensured in procedure *TR-deliver*. In fact, a participant knows at every moment if it has already delivered message m, by storing *m-id(m)*. The algorithm keeps track of the identifiers of garbage collected messages. Furthermore, since we are in an environment devoid of byzantine failures, no spurious messages are delivered.



(a) Unreliable Topic Broadcast

(b) Reliable Topic Broadcast

Figure 9. Throughput of Reliable/Unreliable Topic Broadcast

12. Appendix 3: Detailed performance measurements

We present here the detailed results of our performance measurements summarized in Figure 8(a). Figure 9 shows the performance of both broadcast protocols, together with the variance of the measurements.

In the case of the unreliable broadcast, the variation decreases when the number of participants increases, as conveyed by Figure 9(a). This is due to the fact that the performance is bound by the global performance of the network. In the case of Figure 9(b) in return, the variance remains more stable since the limits of the network are reached very quickly.