Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction*

Patrick Th. Eugster[1], Rachid Guerraoui[1], and Joe Sventek[2]

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

Abstract. Publish/subscribe is considered one of the most important interaction styles for the explosive market of enterprise application integration. Producers publish information on a software bus and consumers subscribe to the information they want to receive from that bus. The decoupling nature of the interaction between the publishers and the subscribers is not only important for enterprise computing products but also for many emerging e-commerce and telecommunication applications.
It is often claimed that object-orientation is inherently incompatible with the publish/subscribe interaction style. This flawed argument is due to the persistent confusion between object-orientation as a modeling discipline and the specific request/reply mechanism promoted by CORBA-like middleware systems. This paper describes object-oriented abstractions for publish/subscribe interaction in the form of Distributed Asynchronous Collections (DACs). DACs are general enough to capture the commonalities of various publish/subscribe interaction styles, and flexible enough to allow the exploitation of the differences between these flavors.

Keywords: Abstraction, concurrency, distribution, asynchrony, publish/subscribe, collection

1 Introduction

This paper presents *Distributed Asynchronous Collections (DACs)*: object-oriented abstractions for expressing different publish/subscribe interaction styles and *qualities of service (QoS)*.

Motivation. With the emergence of wide area networks, the importance of flexible, well-structured and also efficient communication mechanisms is increasing. Basing a complex interaction between multiple hosts on individual *point-to-point* communication models is a burden for the application developer and leads to rather static and limited applications. In mobile communications furthermore, it may not be simple for an application to spot the exact location of a component at any moment. Also may the number of entities interested in certain information vary throughout the entire lifetime of the system. All these constraints visualize

* This work is partially supported by Agilent Laboratories and Lombard Odier & Co.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 252–276, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the demand for more flexible communication models, reflecting the dynamic nature of the applications. The *publish/subscribe* interaction style has proven its ability to fill this gap [22]. Indeed the *decoupling* of parties in *time* as well as *space* is a key to scalability.[1]

Publish/Subscribe Interaction. The classical *topic-based* or *subject-based* publish/subscribe involves a static classification of the messages by introducing group-like notions [26], and is incorporated by most industrial strength solutions, e.g., [7,32]. Publish/subscribe is frequently based on a *push model*, where producers feed a software bus with information and the information is pushed towards consumers from there. Other approaches to "messaging" furthermore integrate *pull-style* mechanisms [24], i.e., consumers actively poll for new information. The term *queueing* is frequently applied when referring to such pull-style interaction. Queueing usually expresses *one-for-all* semantics, which means that one single consumer will consume an information. In contrast, with push-style interaction, an information is generally pushed to all consumers (*one-for-each*). As noticed in [28], some applications need only one interaction style while others require both. Instead of bringing all these variants to a common denominator, much emphasis is usually put on their differences.

Object-Oriented Publish/Subscribe: Does it Make Sense? It is often claimed that "objects" cannot really support the requirements of a publish/subscribe middleware [18]. That argumentation is also commonly used by the promoters of so-called "messaging systems", which claim that objects do communicate through synchronous method invocations which force the interacting parties to be both coupled in time *and* in space.

This paper makes a case against this argument by making an attempt to unify the diverging flavors of publish/subscribe. The argument against a fusion of object-orientation and the publish/subscribe communication style indeed might apply to the current commercial practices in distributed object-oriented computing, which are mainly based on derivatives of the *remote procedure call* (DCOM [25], Java RMI [31], CORBA [23]).[2] As we will convey in this paper, decoupling publishers and subscribers can be made very practical in an object-oriented setting, and the integration of object-oriented principles and messaging can go further than simply wrapping a messaging system with an object-oriented API.

Publish/Subscribe Abstractions. To capture the various interaction styles of publish/subscribe, we propose an 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

[1] 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.

[2] Much effort is currently made to integrate messaging into existing middleware solutions, as shown by [12,24].

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, 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. According to the terminology adopted in the *observer design pattern* [9], the DAC is the *subject* and its client is the *observer*. This abstraction allows to unify different publish/subscribe styles in a single framework, which can be seen as an extension of a conventional collection framework. We will show in this paper how this approach allowed us to mix push and pull models, *one-for-all* and *one-for-each* semantics, along with different *QoS*.

In short, within all publish/subscribe interaction styles none is clearly better than the others for all application purposes. In this paper we present simple abstractions for publish/subscribe interaction, called Distributed Asynchronous Collections. On the one hand, DACs allow to capture the different styles without blurring their respective advantages. On the other hand, DACs unite these styles inside a single framework.

Roadmap. The remainder of this paper is organized as follows. Section 2 recalls the various interaction styles in distributed computing and motivates the need for a subscription-like way of communicating. Section 3 gives an overview of the DAC abstraction. Section 4 gives the basic DAC API, whereas Section 5 presents some preliminary class implementations. In Section 6 we show a simple example of programming with DACs. Section 7 discusses some performance issues of our implementation, and Section 8 contrasts our efforts with related work. Finally Section 9 summarizes our work and concludes the paper.

2 Publish/Subscribe: Commonalities and Variations

Before describing our DAC abstraction, we first overview the basics of publish/subscribe interaction styles. In a first step, the publish/subscribe communication style is compared with more traditional interaction schemes. In a second phase, the different existing approaches to publish/subscribe are elucidated more precisely. We point out the fact that each of the different interaction styles has proven certain advantages over others, which motivates the usefulness of unifying them inside a framework.

2.1 Publish/Subscribe in Perspective

The publish/subscribe paradigm is a loose communication scheme for modeling the interaction between applications in distributed systems. Unlike the classic *request/reply* model or *shared memory* communication, publish/subscribe provides *time decoupling* (i.e., the interacting parties do not need to be up at the same time) of message producers and consumers. Figure 1 shows a comparison of the most common communication schemes: *message passing* (*asynchronous*

send) may also offer an asynchronous interaction scheme, but lacks *space decoupling* (i.e., the interacting parties need to know each other), just like the request/reply communication style. Indeed with message passing, the information producer must have a means of locating the information consumer to which the information will be sent, whereas with the request/reply interaction model the message consumer requires a reference to the information producer in order to issue a request to it. Publish/subscribe combines *time* as well as *space* decoupling, since the information providers and consumers remain anonymous to each other. This outlines the general applicability of this communication model and makes it appealing.[3] Like communication based on shared memory, publish/subscribe moreover allows to address several destinations (*arity of n*). Basically the publish/subscribe terminology defines two roles:

- *Subscriber*: A party which is interested in certain information (events, messages) subscribes to that information, signalling that it wishes to receive all pieces of information (event notifications, messages) manifesting the specified characteristics. *Leasing* is a special form of subscribing, in which the duration of the subscription is limited by a time-out.
- *Publisher*: A party that produces information (events, messages) becomes a *publisher*.

In most applications however, participating entities incorporate both publishers and subscribers, which allows a very flexible interaction. This is one of the main differences to pure *push-based systems* [13], where participants are either producers or consumers and producers are supposed to be several orders of magnitude higher in number than consumers.

	Time	Space	Arity
Request/Reply	Coupled	Coupled	1
Asnynchronous Send	Decoupled	Coupled	1
Shared Memory	Coupled	Decoupled	n
Publish/Subscribe	Decoupled	Decoupled	n

Fig. 1. Different Communication Models

2.2 Topics

The classic publish/subscribe interaction model is based on the notion of *topics* or *subjects*, which basically resemble groups [26]. Subscribing to a topic T can be viewed as becoming member of a group T. The topic abstraction however

[3] It is possible to build closer coupled communication models on top of loose ones and vice versa, as proposed by [34] for instance. The resulting performance in the second case however is generally poor.

differs from the group abstraction by its more dynamic nature. While groups are usually disjoint sets of members (e.g., group communication for replication [3]), topics typically overlap, i.e., a participant subscribes to more than just one topic. In order to classify the topics more easily, it is of great use to furthermore introduce a hierarchy of topics [32]. In this model, a topic can be a derived or more specialized topic of another one, and is therefore called *subtopic*. The use of wildcards offers a more convenient way of expressing *cross-topic* requests.

Figure 2 shows an example of topic-based subscribing. Subscriber S_1has announced its interest in both topics x and y. It is notified of events corresponding to both topics (messages m_x and m_y). Subscriber S_2 has only subscribed to topic x, and therefore only receives messages related to that topic (message m_x).

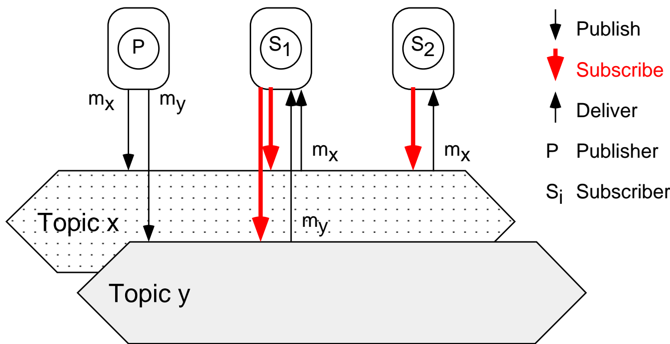

Fig. 2. Topic-Based Subscribing

2.3 Push and Pull Mixing

In the publish/subscribe model, the action of subscribing describes a sort of registration procedure for an interested party. However, interests in events can also be expressed through a more direct interaction. In general, we distinguish the following ways for an interested party to interact:

– In a passive way, it can subscribe to a choice of notifications. By callbacks it will be notified of the occurrence of events. This kind of interaction constitutes the *push model*, since the information is pushed from the information bus to the subscriber. This is the classic publish/subscribe approach, since it enforces applications which are only loosely coupled in *time*.
– More actively, a consumer can *poll* for new notifications. This task may waste resources and is not well adapted to asynchronous systems. In fact, polling based solutions tend to be very expensive and scale poorly: polling too often can be inefficient and polling too slowly may result in delayed responses to critical situations [29]. This style is often called *queueing*, and in the context of this paper this is the type of interaction we will refer to as *pull model*.

– Another yet more synchronous pull-type interaction is given by *blocking* pull-style interaction. In this scenario, a consumer which tries to pull information is blocked until a new notification is available. Just like the request/reply model however, this variant introduces time coupling, and is therefore rarely used in common messaging systems.

Although in general push-style interaction seems more appropriate, certain applications may not be interested in receiving information as soon as possible, but only at precise moments. In those situations, the pull model might be of interest.

2.4 Delivery Semantics and Reliability Issues

In distributed systems, and in particular when considering communication models and protocols, precise specification of the semantics of a delivery is a crucial issue. Delivery guarantees are often limited by the behavior of deeper communication layers, down to the properties of the network itself, limiting the choice of feasible semantics. On the other hand, different applications also may demand for different semantics. While sometimes a high throughput is preeminent and a low reliability degree is tolerable, some applications prioritize reliability to throughput. For this reason most common messaging systems provide different *qualities of service*, in order to meet the demands of a variety of application purposes [1,32].[4] The delivery semantics for notifications offered by existing systems can be roughly divided into two groups.

– *Unreliable delivery.* Protocols for unreliable delivery give few guarantees. These semantics are often used for applications where the throughput is of primary importance, but the loss of certain messages is not fatal for the application.
– *Reliable delivery.* Reliable delivery means that a message will be delivered to every subscriber despite certain failures. Usually the failure or the absence of the subscriber itself is not considered, i.e., if the subscriber has failed, the message might not be delivered to it and the reliability property is not considered violated. When using persistent storage to buffer such messages until the subscriber is back on line, a stronger guarantee is given. This is often referred to as *certified delivery* [32].

3 Distributed Asynchronous Collections: Overview

This section gives an overview of our approach to publish/subscribe, by first introducing *Distributed Asynchronous Collections* as key abstractions. We show the relationship between those abstractions and the publish/subscribe communication model. In a second step, we picture more in detail how these abstractions allow to build several different publish/subscribe variants inside a unified framework. This section however should be understood as a general introduction to

[4] [32] adopts the notion of *delivery service.*

our abstractions for publish/subscribe. The following sections will give a more concrete view of DACs.

3.1 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, like a mail folder or a file directory. Unlike a conventional collection, a DAC is a distributed collection whose operations might be invoked from various nodes of a network. DACs differ fundamentally from the distributed collections described in [21] for instance, by being asynchronous and essentially distributed, i.e., DACs can be seen as omnipresent entities.[5] Participating processes act with a DAC through a local proxy, which is viewed as a local collection and hides the distribution of the DAC. DACs are not centralized on a single host, in order to guarantee their availability despite certain failures.

A collection framework is a unified architecture for representing and accessing collections, allowing them to be manipulated independently of their representation. For example, both Smalltalk [14] and Java [16] contain rich collection frameworks that reduce the programming effort by providing useful data structures and algorithms together with high-performance implementations. Collection frameworks can for instance also be found for C++ (e.g., Silicon Graphics' STL [30]) as additional libraries. Figure 3 shows the inheritance graph of the Java collection framework.

3.2 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 distant object can involve a considerable latency, hardly comparable with that of a local one. In contrast, asynchronous interaction is enforced with our collections. By calling an operation of a DAC, one expresses an interest in *future notifications*. When querying a DAC for objects of a certain kind for instance, 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.

There is a strong resemblance with the notion of *future* [4] (*future type message passing* [35]), that describes a communication model in which a client queries an *asynchronous object* for information by issuing a request to it. Instead of blocking however, the client can pursue its processing. As soon as the reply has been computed, the object acting as server notifies the client. Latter one may query the result (*lazy synchronization* or *wait-by-necessity* [5]), or ignore it. Figure 4 compares the two paradigms. When programming with DACs, the

[5] The distributed collections presented in [21] are centralized collections that can be remotely accessed through RMI.

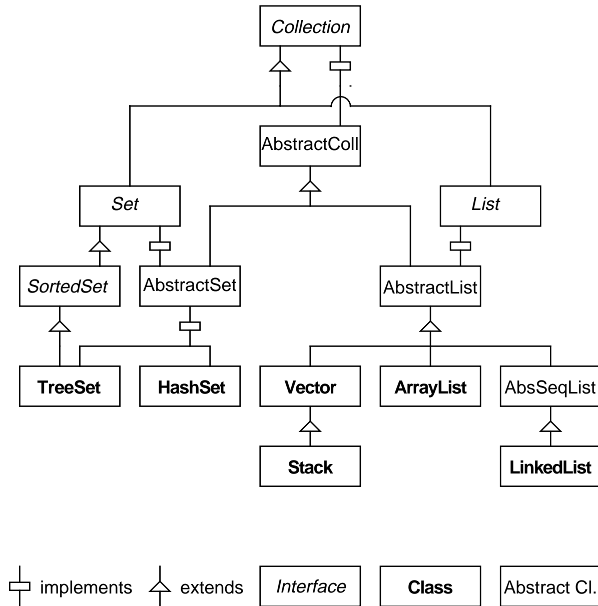

Fig. 3. Collections in Java (Excerpt)

subscriber can be viewed as the client. The DAC incarnates a server role in this scenario, since the publishers, which are the effective information suppliers, remain anonymous.

By calling an operation on the DAC, the caller requests certain information. The main difference with futures lies in the number of times that information is supplied to the client. Within the notion of future, only a single reply is passed to the client,[6] whereas with DACs, every time an information which is interesting for the registered party is created, it will be sent to it.

3.3 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. It is always possible to insert a new element into a DAC. In the sense of publish/subscribe, inserting an object

[6] ABCL/1 represents an exception, in the sense that several replies may be returned [35].

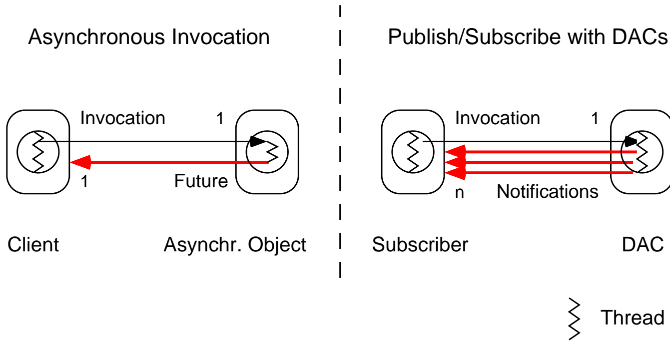

Fig. 4. DACs vs. Future

into a DAC also means to publish that object for the topic represented by the DAC. Every DAC can thus be viewed as a publish/subscribe engine of its own. Figure 5 shows the traditional topic-based publish/subscribe scheme. The topic is represented by an attribute of the message, and the application has to deal with it explicitly. Since a DAC is bound to a topic, the topic is given implicitly, and appears only in the protocol message which is hidden from the application, as shown in Figure 6. It encapsulates the application message.

Existing publish/subscribe frameworks introduce specialized message types, e.g., [12]. 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 of any kind of object. In Java, this is expressed by allowing any object of class `java.lang.Object` to be passed as a message.[7]

Message m	`public class Message {`
	`  public String topic;`
	`  public String content;`
	`}`

Criteria	topic of m is “/Chat/Insomnia”
Argument	`String topic = "/Chat/Insomnia"`
Evaluation	`m.topic.equals("/Chat/Insomnia")`
Deliver	`m`

Fig. 5. “Traditional” Topic-Based Publish/Subscribe

[7] In order to be conveyable, a Java object should furthermore implement the `java.io.Serializable` interface [15], which contains no methods.

Protocol Message p	`public class Message {`
	`public String getTopic() {...}`
	`public Object getMsg() {...}`
	`...`
	`}`

Message m	`public class ChatMsg {...}`
Criteria	topic of m is "/Chat/Insomnia"
Argument	`String topic = "/Chat/Insomnia"`
Evaluation	`p.getTopic().equals("/Chat/Insomnia")`
Deliver	`m = p.getMsg()`

Fig. 6. Topic-Based Publish/Subscribe with DACs

4 DAC Interfaces

The previous section introduced DACs as general abstractions for publish/subscribe. This section presents the main interfaces of our DAC realization in Java. In the context of this paper, we will limit ourselves to describing the functionalities which are common to all DAC subinterfaces, in order to show their similarity to operations on conventional centralized collections.

4.1 Topic-Based Subscribing with DACs

In our system, each *topic* is represented by a DAC, and is denoted by a name, like "Chat". A DAC constructor thus requires an argument which denotes the name of the topic it will represent (see Section 6 for an example). Topics can have specializations, or *subtopics*, and connecting to a topic requires the name in a URL-type format. Typically, "/Chat/Insomnia" is a reference to the topic called "Insomnia" which is a subtopic of "Chat". The root of the hierarchy is represented by an *abstract* topic (denoted by "/"). Top-level topics, which are no specializations of already existing ones, are subtopics of the abstract root topic only. Subscribing to a topic can trigger subscriptions for the subtopics as well, as illustrated in Figure 8. Subscriber S_1 subscribes to topic "Chat" 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 "/Chat/Insomnia". In contrast, S_2 only subscribes to "/Chat/Insomnia" and thus does not receive message m_2, which belongs to the *supertopic*. With the *push* model adopted in DACs, subscribing entities must register a *callback* object. That callback object must implement a specific interface, namely the `Notifiable` interface, shown in Figure 7. Through a call to the `contains()` method, the DAC notifies the subscriber that it contains a new notification. The second argument enables the use of the same callback object for several topics.

```
public interface Notifiable {

  public void contains(Object msg, String topicName);
}
```

Fig. 7. Interface Notifiable

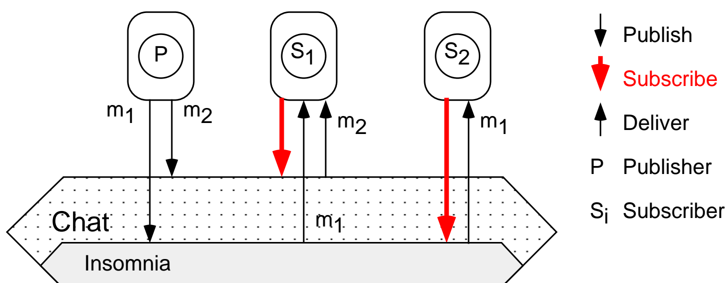

Fig. 8. Topic-Based Publish/Subscribe with DACs

4.2 DAC Methods

Figure 9 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.

Synchronous Methods. Since a *DAC* is in the first place a collection, the `DAC` interface inherits from the standard `java.util.Collection` interface. The inherited methods are not denatured but adapted, and we denote them as *synchronous.*

- `get()`. Similarly to a centralized collection, calling this method allows to retrieve objects. The synchronization it introduces is however very weak, since it returns `null` in the absence of unconsumed information, as explained in Section 2. Which element will be returned depends on the nature of the collection (see Section 5 for more details). This implements the *pull* model.
- `contains()`. A DAC is first of all a representation of a collection of elements. This method allows to query the collection for the presence of an object. Note that an object that is contained in a DAC belongs to the topic represented by that DAC.
- `add()`. This method allows to add 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. An asynchronous variant of this method could consist in advertising the eventual production of notifications.

This could furthermore be combined with the registration of a callback object, that the DAC would poll in order to obtain new event notifications. In the terminology adopted in [24], this is called a *pullsupplier*.

Asynchronous Methods. We have added several *asynchronous* methods to express the decoupled nature of publish/subscribe interaction specific to DACs. In these methods, asynchrony is expressed by an additional argument, denoting a callback object which implements the `Notifiable` interface. Not all operations known from conventional collections find an analogous meaning in an asynchronous distributed context, and our ongoing research in that domain might cause minor modifications to this interface.

– `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 should be eventually pushed into the collection. The interested party advertises its interest by providing a reference to an object implementing the `Notifiable` interface (Figure 7), through which it will be notified of events.
– `containsAll(Notifiable n)`. This method offers the same signature than the previous method. The difference is that a subscription is generated for all subtopics of the topic represented by this DAC. This reflects the situation given in Figure 8.
– `remove(Notifiable n)`. Likewise, by calling one of these methods, 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* [32] in contrast to *one-for-each*,[8] implemented by the asynchronous `contains()` methods, where a message is sent to all.
– `removeAll(Notifiable n)`. This method is similar to the previous one, except that a subscription is generated for all subtopics of the topic represented by this DAC.
– `clear(Notifiable n)`. The conventional argument-less `clear()` method allows to erase all elements from the collection, whereas this asynchronous variant expresses the action of *unsubscribing*.

5 DAC Classes

The previous section focused on the interfaces, through which an application can use DACs in order to benefit from the strength of our publish/subscribe abstractions. As depicted earlier, our framework consists of a variety of DACs spanning

[8] By using the formalism of [27], one could say that *every Nth occurrence* of an event is notified to a subscriber, with N being the total number of subscribers, and no event being delivered to more than one subscriber.

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

{
  public Object get();
  public boolean contains(Object message);
  public boolean add(Object message);
  ...
  public boolean contains(Notifiable N);
  public boolean containsAll(Notifiable N);
  ...
  public boolean remove(Notifiable N);
  public boolean removeAll(Notifiable N);
  ...
  public void clear(Notifiable N);
  ...
}
```

Fig. 9. Interface DAC (Excerpt)

different semantics and guarantees, since different applications have different requirements. These semantics can be seen as different *QoS*. While certain properties of DACs reflect in their interfaces, certain semantics do not appear in the API. These parameters influence the classes implementing those interfaces, and thus lead to a variety of classes implementing the same interface. This section presents the different properties of the classes constituting our framework.

5.1 Delivery Semantics

When a producer publishes a message, it does not directly interact with subscribers. To whom exactly the message will be delivered does not show in the DACs interface. Parts of the semantics do not come to light in the interfaces. The underlying multicast protocols 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`). By distinguishing between unreliable and reliable DACs our framework hierarchy is roughly split into two subtrees, as shown in Figure 10.

5.2 Duplicates

Just like it is possible to have duplicate elements in centralized collections, it is possible in Distributed Asynchronous Collections that a same message is delivered more than once. In fact, the two are closely related in our context. If a DAC does not accept duplicates, it should not deliver any duplicates to subscribers.

The simple `DAWeakBag` class for instance does not prevent a notification to be delivered more than once, whereas the `DAWeakSet` class gives stronger guarantees by eliminating duplicate elements. This property is orthogonal to other characteristics of our DACs. For that reason, our framework contains a variant with and without duplicates for every other property, as shown in Figure 10. When allowing duplicates and combining with unreliable delivery for instance, the outcome is *best-effort* semantics. In return, with reliable delivery, *at-least-once* semantics can be guaranteed.

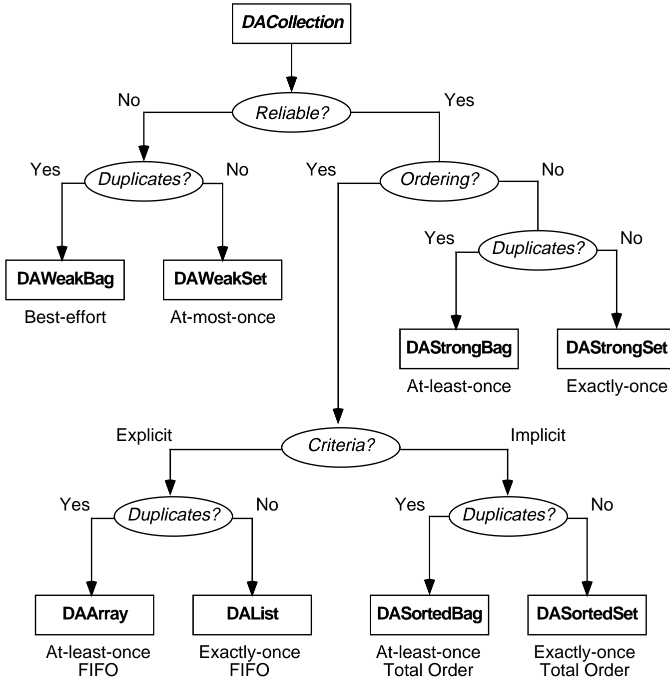

Fig. 10. DAC Framework

5.3 Storage vs. Delivery Order

Collections are often characterized by the way they store their elements. *Sets* or *bags* for instance do not rely on a deterministic order of their elements. Conversely, *sequences* can store their elements in an order given explicitly or implicitly based on properties of the elements. In Distributed Asynchronous Collections 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 DACs, an implicit order is a global delivery order on which the DAC itself decides. The `DASortedSet` class for instance presents a *total order* of 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.

5.4 Insertion Order

In different centralized collections, the insertion order may have an impact on the storage order. In a *queue* or a *stack* for instance, the chronological insertion order will drive the storage order as well as the extraction order. A position can be given as additional argument to an insertion into a *list* for instance. In an asynchronous collection however, the order of insertion corresponds to the order of sending or 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, since inserting a message at the beginning of a list would translate to sending a message before messages that have possibly already been delivered to subscribers. Therefore there is never any explicit argument for the order passed when "inserting" a new element into a DAC.

5.5 Extraction Order

Extracting an element from a centralized implementation corresponds to pulling messages from a distributed asynchronous one. In the case of consumers polling a DAC for new messages, two different policies may be applied:

– *FIFO.* The collection behaves like a queue by returning the first received and undelivered message. In fact, the DAC proxy contains a buffer, in which received messages are inserted. From there, they are delivered to the pulling consumer in a FIFO order.
– *LIFO.* The collection acts like a stack and delivers the latest received message. The principle is the same than above, except that the messages are delivered in a LIFO order from the buffer to the consumer.

Therefore when using a pull model, the application has the choice between queues and stacks. Any class presented in Figure 10 can be used both as stack or queue.

Messages may be volatile, which means that they may be dropped immediately after delivery. Conversely, the message could be stored in memory or even on persistent storage. In the context of this work however, we did not deal with message storage so far. Messages are considered volatile, and are dropped as soon as they have been consumed. Missed messages are therefore not replayed to *late subscribers* or temporarily disconnected participants.

6 Putting DACs to Work

In this section we describe a simple example application using the flexibility of Distributed Asynchronous Collections. It shows how to implement *chat sessions* based on simple DACs.

We will concentrate on two users, Alice and Tom. They are both chat addicts, and love to chat deep into the night. Therefore they subscribe to the topic "Insomnia" which is a subtopic of "Chat" to receive all messages from like-minded chatters (see Figure 11). For the sake of simplicity, we will assume that this evening Tom is missing inspiration, and therefore takes a pure subscriber role. Alice on the other hand, is very talkative, and publishes several messages. Figure 12 shows class `ChatMsg`, which represents a possible message class for this application.

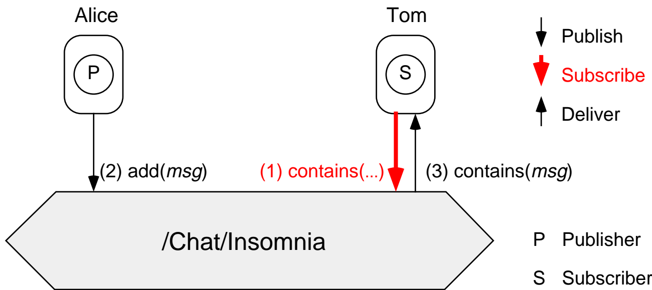

Fig. 11. Chatters

```
public class ChatMsg
  implements java.io.Serializable

{
  private String sender;
  private String text;
  public String getSender() { return sender; }
  public String getText() { return text; }
  public ChatMsg(String sender, String text) {
    this.sender = sender; this.text = text; }
}
```

Fig. 12. Event Class for Chat Example

6.1 Publishing for a Topic

When making use of topic-based publish/subscribe, a topic is represented by a DAC, as seen previously. In order to access a DAC from a process, a proxy must be created. This requires an argument denoting the name of the topic it bears. Except for that argument, the action of creating a proxy is indistinguishable from creating a local collection. The DAC instance called *mychat* in Figure 13 henceforth allows us to access the topic "/Chat/Insomnia". Now it is possible to directly publish and receive messages for the topic associated to that DAC.

Creating an event notification for a topic consists in inserting a message object into the DAC by issuing a call to the `add()` method (see Section 4), from where it is accessible for any party. It is more favorable for consumers to be notified automatically when a new message has been published, than to waste computation time on polling activity. For that purpose, a party interested in a topic can register as subscriber.

```
DASet sleeplessChatters = new DAStrongSet("/Chat/Insomnia");
String me = "Alice";
ChatMsg msg = new ChatMsg(me, "Hi everyone");
sleeplessChatters.add(msg);
```

Fig. 13. Publishing a Message

6.2 Subscribing to a Topic

In order to subscribe to a topic an interested party must provide a callback object implementing the `Notifiable` interface. The callback method comprises two arguments. The first argument represents the effective message, and the second argument represents the name of the topic the message was published for. This provides more flexibility, since the same subscriber object can be used to receive messages related to several topics. In the above example, a subscriber may be interested in all ongoing chat sessions, and not only in "Insomnia". Figure 11 shows the interactions with the DAC, and Figure 14 shows the corresponding code that allows Tom to subscribe.

7 Implementation Issues

This section discusses the realization of our first DAC implementation, including first performance measurements. We draw preliminary conclusions of our prototype, which has been developed in pure Java and relies on UDP, thus increasing its portability.

```
class ChatSubscriber
  implements Notifiable

{
  public void contains(Object msg, String topic) {
    System.out.println(((ChatMsg)msg).getText());
  }
}

DASet sleeplessChatters = new DAStrongSet("/Chat/Insomnia");
Notifiable sub = new ChatSubscriber();
sleeplessChatters.contains(sub);
```

Fig. 14. Topic-Based Publish/Subscribe with DACs

7.1 Inside DACs

The effective DAC class as it is perceived by the application only represents a small portion of the underlying code. Redundant code has been avoided by a modular design and using inheritance. Figure 15 shows the different layers in our implementation. These layers do not necessarily correspond to Java classes, but represent protocol layers.

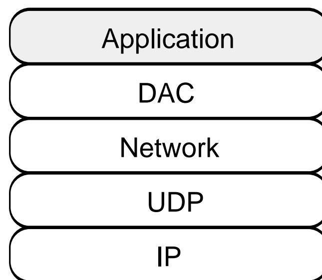

Fig. 15. Layers

– **The DAC layer.** This layer is composed of the classes implementing directly the DAC interfaces. They are rather lightweight classes, which delegate general functionality to the underlying layer. Their tasks are similar to centralized container classes. They mainly take care of the local management of messages, and furthermore handle the subscriptions. The most frequent interaction model is the callback model (push-model), where subscribers do not poll for new messages but are called back upon incoming messages. In

that case the DAC applies a predefined threading model, by assigning notifications to threads.

- **The Network layer.** The Network layer regroups common functionalities of all DACs, like publishing messages or forwarding subscription information. It hides any remote party involved in same topics from the DAC layer. This layer maintains a form of network topology knowledge, which basically consists of its immediate neighbors.
- **The UDP layer.** Our entire publish/subscribe architecture is finally implemented on top of UDP. UDP is a non reliable protocol, which offers us the looseness required for the decoupled nature of publish/subscribe. Java offers classes for UDP sockets and datagrams (`java.net.DatagramPacket` and `DatagramSocket`), which are pretty close to the metal.

7.2 Performance

The performance tests of our prototype were made on HP workstations running HP-UX 10.20 and JVM 1.1.5 and 1.1.6. on a normal working day. The implementation uses a marshalling/unmarshalling procedure built from scratch and optimized for each event type (the Java serialization classes were not used, since they are usually considered rather slow). Four example message types were considered:

- `Integer.` This corresponds to the basic Java `int` type.
- `String.` Java type `String` with a length varying between 10 and 20
- `DetailRecord.` This is a class containing four attributes, of which two represent dates (Java type `Date`) and two are strings (Java type `String`).
- `CallDetailRecord.` A subtype of `DetailRecord`. In addition to the attributes of the latter one, a `CallDetailRecord` furthermore contains four integers and two strings.

In our measurement scenario, several subscribers asynchronously receive events for a topic where a publisher produced the events. The numbers of messages considered for a single run of the experiment varied between 10 and 1000 and the measures obtained conveyed an average result after several experiments of the same profile.

Figure 16 shows the latency when publishing. For example, a publisher needs 3s to publish 100 events of type `DetailRecord`. They include the time for marshalling each of the events and the time to put the events into the UDP socket.

Figure 17 shows the global throughput for the same scenario. It takes for instance 5s until a subscriber has received 100 events of type `DetailRecord`. The 5s correspond therefore to the time spent at the publisher side and the subscriber side of the DAC. They include the time for marshalling, remote communication and unmarshalling.

These simple measurements allowed us to do draw several preliminary conclusions:

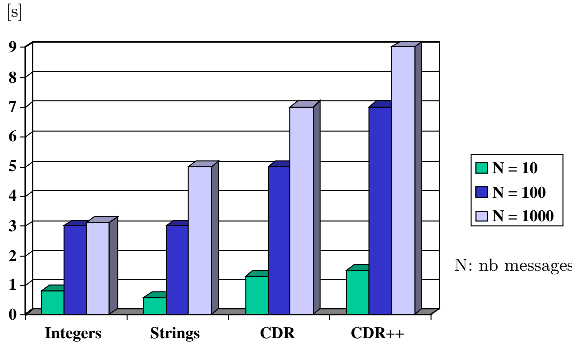

Fig. 16. Latency

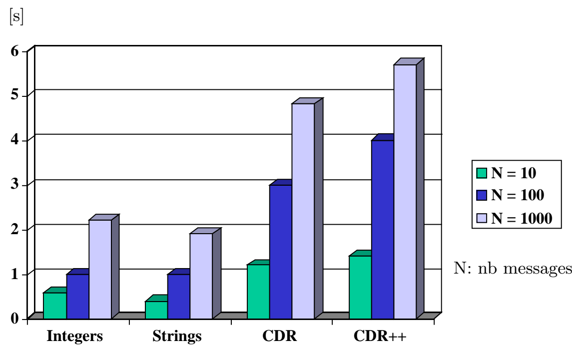

Fig. 17. Throughput

– The complexity of the event type has a heavier impact on the time it takes for a publisher to send events then on a subscriber to receive events. This is not surprising because in the first case, the marshalling time is more significant (there is no inherent cost of remote communication).
– It might look surprising that integers take longer than strings. In this implementation however, everything is converted to strings in the serialization procedure.
– Finally, the overall measures confirm the very fact that nowadays, optimizing marshalling is at least as important as optimizing remote communication.

8 Related Work

During the last years, the need for large scale event notification mechanisms has been recognized. Much effort has therefore been invested in this domain, and a multitude of approaches have emerged from academic as well as industrial

impulses. We present here the main characteristics of related approaches and we compare them with our Distributed Asynchronous Collections.

8.1 Event Service Specifications

In order to integrate the publish/subscribe communication style into existing middleware standards, specifications have been conceived by both the Object Management Group [24] and Sun [12,2,6].

The OMG has specified a CORBA service for publish/subscribe oriented communication, called the *CORBA Event Service*. The specification is aimed to be general enough to not preclude sub-specifications and various implementations that would match the needs of specific applications. According to the general service specified however, a consumer subscribes to a channel expressing thereby an interest in receiving *all* the events from the channel. In other words, filtering of events is done according to the channel names, which basically correspond to topic names. When the consumer subscribes to the channel, it is supposed to receive all events put in the channel. Event channels are CORBA objects themselves, and in current implementations they are centralized components. Therefore these engines manifest a strong sensitivity to any component failure, which makes them unsuitable for critical applications.

The *Java Messaging Service* [12] 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 goal to merely support the JMS API directly. Its generic nature, required in order to conform to a maximum number of existing systems, appears to be rather cumbersome. Applications are very aware of the underlying messaging service, and developers must make themselves acquainted with the API.

The *Java Distributed Event Specification* [2] explicitly introduces the notion of event *kind*. Registration of interest indicates the kind of events that is of interest, while a notification indicates an occurrence of that kind of event. One can combine this notion with that of *JavaSpace* [8] to provide support for topic-based publish/subscribe notification. Inspired by Linda [10], a JavaSpace is for example a container of objects that might be shared among various suppliers and consumers. The JavaSpace type is described by a set of operations among which a *read* operation to get a copy of an object from a JavaSpace, and a *notify* operation aimed at alerting some potential consumer object about the presence of some specific object in the JavaSpace. Combined with the Java Distributed Event interfaces, one can build a publish/subscribe communication scheme where a JavaSpace plays the role of the event channel aimed at broadcasting events (notifications) to a set of subscriber objects. The nature of the subscription is however not specified and it is not clear whether one would be able to subscribe to a particular operation.

The *InfoBus 1.2 Specification* [6] describes an information bus which enables dynamic data exchange between JavaBeans. Components must implement a minimal interface in order to plug into the bus. As a member of the bus any

component can exchange data structured as arrays, tables, or database rowsets with other components. Interestingly, adapted collection types are available for InfoBus, which ease the transfer of collections of objects.

These standards are based on specifications and it would be interesting to see how one could implement services that comply with these standards using DACs. Note however that the CORBA Event Service does not present any hierarchical arrangement of channels, and the Java Messaging Service introduces explicit message classes which require explicit marshalling/demarshalling.

8.2 Established Systems

Most industrial strength solutions involve topic-based publish/subscribe. *Smartsockets* [7] or *TIB/Rendezvous* [32] are such engines.

In Smartsockets, an event channel can accept subscriptions for specific topics. A consumer receives all the event notifications that belong to the topic to which it has subscribed. The topic defines a kind of virtual connector between objects of interest and recipients. If a producer is interested in producing an event on a number of topics or channels, it has to explicitly publish the event on all of them. Event notifications are represented by records, nevertheless custom event types may be defined.

A similar approach was adopted in the development of the TIB/Rendezvous infrastructure. A hierarchical naming model corresponds to the hierarchical organization of the entities of interest. Just as Uniform Resource Locators (URLs) provide a way of locating and accessing Internet resources, a naming scheme is provided to locate and access events of interest. The naming scheme proposed can use wildcards, which allows to subscribe to patterns of topics. TIB/Rendezvous provides a certain degree of fault-tolerance, and makes usage of IP-multicast. Event notifications are composed of a set of typed data fields, including the topic.

Most industrial systems implement API's for object-oriented languages, like the JMS specification for Java. These solutions however did not undergo a fundamentally object-oriented design, but only offer an object-oriented layer on top of a messaging system.

8.3 Collections

Both Java and Smalltalk offer integrated collection frameworks. These only span the most common collection types. More specific collections can be found as external libraries. For instance, *JGL* [21] and the `util.concurrent` [20] package offer more elaborate collection types for Java.

JGL is a first approach to distributed collections in Java. It 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, covering only the main features used by the majority of Java developers. 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.

The `util.concurrent` package provides the application programmer with a set of collections especially targeted at resolving concurrency problems. It contains for instance collections which alleviate concurrent traversals by making each time a copy of the array backing the collection. Another feature are synchronization wrappers for standard collections, with the possibility to specify external read and/or write locks.

In contrast to JGL, our DACs avoid any single point of failure and are essentially distributed. To exploit this distribution, asynchronous interaction is enforced. Synchronization as discussed in the context of the `util.concurrent` package is an issue we do not address with our DACs, but could be the topic of future work.

9 Concluding Remarks

It has long been argued that distribution is an implementation issue and that the very well known metaphor of objects as "autonomous entities communicating via message passing" can directly represent the interacting entities of a distributed system. This approach has been conducted by the legitimate desire to provide distribution transparency, i.e., hiding all aspects related to distribution under traditional centralized constructs. One could then reuse, in a distributed context, a centralized program that was designed and implemented without distribution in mind.

As argued in [33,19,11] however, distribution transparency is a myth that is both missleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction. 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 usually expensive and hard, if not impossible to implement in the presence of network failures (*partitions*).

We have been considering an alternative approach where the programmer would be very aware of distribution but where the ugly and complicated aspects of distribution would be encapsulated inside specific abstractions with a well-defined interface. This paper presents a candidate for such an abstraction: the *Distributed Asynchronous Collection*. It is a simple extension of the well-known collection abstraction. DACs add an asynchronous and distributed flavor to traditional collections [4], and enable to express various forms of publish/subscribe interaction. In fact, most systems we know about are unwieldy and consider only a limited set of interaction models. DACs are general lightweight publish/subscribe abstractions: they can be introduced through a library approach and they allow to express various interaction types and QoS.

We believe that our object-oriented view of publish/subscribe is a unique compromise between transparency and efficiency. By offering a modular design aligned with different communication semantics, we enforce ease of use without

missing performance related issues. We are currently making use of DACs in various practical examples, which are far more complex than the simple chat example given in this paper. The objective of investing in several applications is to end up with a stable framework, that would for instance extend JGL. The issue of translating operations known from conventional collections to an asynchronous distributed context is however not entirely completed, and certain parts of the API might be affected by future modifications. We also explore approaches to express *content-based* publish/subscribe and specific algorithms to realize efficient matching. This is especially challenging in a mobile environment, where nodes might be disconnected, and objects might migrate from a node to another [17].

References

1. M. Altherr and M. Erzberger and S. Maffeis. iBus - A Software Bus Middleware for the Java Platform. In International Workshop on Reliable Middleware Systems, pages 43-53, October 1999.
2. K. Arnold and B. O'Sullivan and R.W. Scheifler and J. Waldo and J. Wollrath. The Jini Specification. Addison Wesley, 1999.
3. K.P. Birman. The Process Group Approach to Reliable Distributed Computing. In Communications of the ACM, 36:12, pages 36-53, December 1993.
4. J.-P. Briot and R. Guerraoui and K.-P. Löhr. Concurrency and Distribution in Object-Oriented Programming. In ACM Computing Surveys, September 1998.
5. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. In Communications of the ACM, vol. 36, pages 90-102, September 1993.
6. M. Colan. InfoBus 1.2 Specificationa. Sun Microsystems Inc., February 1999.
7. Talarian Corporation. Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper). http://www.talarian.com/, 1999.
8. E. Freeman and S. Hupfer and K. Arnold. JavaSpaces Principles, Patterns, and Practice. Addison Wesley, 1999.
9. E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
10. D. Gelernter. Generative Communication in Linda. In ACM Transactions on Programming Languages and Systems, 7:1, pages 80-112, Januaray 1985.
11. R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. In Informatik, 2, April 1999.
12. M. Happner and R. Burridge and R. Sharma. Java Message Service. Sun Microsystems Inc., October 1998.
13. M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. In ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), September 1999.
14. IBM. Smalltalk Tutorial. http://www.smalltalksystems.com/references.htm/, 1995.
15. Sun Microsystems Inc. The Java Platform 1.2 API Specification. http://java.sun.com/products/jdk/1.2/, 1999.
16. Sun Microsystems Inc. The Java Collections Framework. http://java.sun.com/products/jdk/1.2/, 1999.

17. E. Jul and H. Levy and N. Hutchinson and A. Black. Fine-grained mobility in the Emerald System. In ACM Transactions on Computer Systems, 6:1, pages 109-133, February 1988.
18. P. Koenig,. Messages vs. Objects for Application Integration. In Distributed Computing, 2:3, pages 44-45, April 1999, BCI.
19. 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/.
20. D. Lea. Overview of package util.concurrent Release 1.2.5. http://gee.cs.oswego.edu/dl/classes/, October 1999.
21. ObjectSpace. JGL - Generic Collection Library. http://www.objectspace.com/products/jgl/, 1999.
22. B. Oki and M. Pfluegl and A. Siegel and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In Fourteenth ACM Symposium on Operating System Principles, pages 58-68, December 1993.
23. OMG. The Common Object Request Broker: Architecture and Specification. February 1998.
24. OMG. CORBAservices: Common Object Services Specification. December 1998.
25. Microsoft Co. DCOM Technical Overview (White Paper), 1999.
26. D. Powell. Group Communications. In Communications of the ACM, 39:4, pages 50-97, April 1996.
27. 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.
28. D. Schmidt and S. Vinoski. Overcoming Drawbacks in the OMG Event Service. In SIGS C++ Report magazine, 10, June 1997.
29. D. Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. http://www.vitria.com, 1998.
30. A. Stepanov and M. Lee. The Standard Template Library. Silicon Graphics Inc., October 1995.
31. Sun Microsystems Inc. Java Remote Method Invocation - Distributed Computing for Java (White Paper). http://java.sun.com/marketing/collateral/javarmi.html/, 1999.
32. TIBCO Inc. TIB/Rendezvous White Paper. http://www.rv.tibco.com/whitepaper.html, 1999.
33. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. A Note on Distributed Computing. Sun Microsystems Inc., November 1994.
34. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. Events in an RPC Based Distributed System. Sun Microsystems Laboratories Inc., November 1995.
35. A. Yonezawa and E. Shibayama and T. Takada and Y. Honda. Object-Oriented Concurrent Programming. In Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, pages 55-89, MIT Press, 1987.