Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction*

Patrick Eugster † Rachid Guerraoui† Joe Sventek‡
†Swiss Federal Institute of Technology, Lausanne
‡Agilent Laboratories, 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 styles, and flexible enough to allow the exploitation of the differences between these styles.

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

1 Introduction

This paper presents *Distributed Asynchronous Collections (DACs)*: object-oriented abstractions for expressing the many diverging publish/subscribe interaction styles.

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 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 [OPSS93]. Indeed the *decoupling* of parties in *time* as well as *space* is a key to scalability.[1]

Publish/Subscribe Dialects. There are different established variants of the publish/subscribe interaction model, each one presenting its respective advantages but also shortcomings. The classical *topic-based* or *subject-based* style involves a static classification of the messages by introducing group-like notions [Pow96], and is incorporated by most industrial strength solutions, e.g., [Cor99, TIB99]. However, research efforts have been targeted more towards *content-based* publish/subscribe [CRW98, SA97, BCM+99].[2] This more flexible variant 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. When disseminating messages over wide area networks, this variant in return requires a fine tuned filtering strategy, in order to avoid a flood of needless notifications and subscriptions. While classical publish/subscribe is based on a *push model*, some approaches to "messaging" furthermore integrate *pull-style* mechanisms [OMG98]. As noticed in [SV97] in fact, some applications need only one interaction style while others require both. Instead of bringing all these variants to a com-

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

[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] The taxonomy introduced in [RW97] refers to this as *property-based*.

mon 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 [Koe99]. The flawed rational under that argument is twofold. First, and this is the argument commonly used by the promoters of so-called "messaging systems", it is claimed that objects do communicate through synchronous method invocations which force the interacting parties to be both coupled in time *and* in space. Second, effective filtering in content-based publish/subscribe requires expressing properties of objects and that usually violates object encapsulation.

This paper makes a case against these arguments and furthermore attempts to unify the diverging variants of publish/subscribe. The first 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 synchronous remote method invocations (DCOM, Java RMI, CORBA).[3] As we will convey in this paper, decoupling publishers and subscribers can be made very practical in an object-oriented setting. The second argument might apply to current implementations of content-based systems. When applying the same principles to an object-oriented environment, one might end up in the situation where the encapsulation property is violated by exposing the state of objects through a query language used for the *subscription*. Again, and as we will show in the paper, content-based filtering can indeed be implemented in a way that completely preserves encapsulation, namely by using *reflection* [KdRB91].

Publish/Subscribe Abstractions. To capture the various types 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 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. According to the terminology adopted in the *observer design pattern* [GHJV95], 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 different publish/subscribe variants together with push and pull models, *one-for-all* and *one-for-each* semantics, along with different *qualities of service*. Besides showing how collections provide a natural way of expressing topic-based publish/subscribe, we illustrate how reflection allows to realize content-based publish/subscribe in a manner that does not violate encapsulation.

In addition we introduce in this paper the notion of *type-based* publish/subscribe, which is a new variant of the publish/subscribe communication model: instead of explicitly associating an event to a topic, the *type* is used in a natural way to categorize the event. In other terms, the notion of *event kind* is matched with that of *event type*. Our framework furthermore allows to combine type-based publish/subscribe with content-based subscription facilities, the same way it provides for an original marriage of topic-based with content-based publish/subscribe.

In short, within all publish/subscribe models 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 discuss some performance issues of our implementation, and Section 7 contrasts our efforts with related work. Finally Section 8 summarizes our work and concludes the paper.

Appendix A contains excerpts of the API and two classes illustrating our approach to content-based publish/subscribe. Appendix B shows step by step how to put DACs to work through a small example application, followed by the complete code for that example.

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

[3] Much effort is currently made to integrate messaging into existing middleware solutions, as shown by [HBS98, OMG98].

the fact that each of the different variants 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* (*singleton 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.[4] Like communication based on shared memory, publish/subscribe moreover allows to address several destinations (*arity of n*). Basically the publish/subscribe terminology defines two players:

- *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* [HJ99], where participants are either producers or consumers and producers are supposed to be higher in number than consumers.

[4] It is possible to build closer coupled communication models on top of loose ones and vice versa, as proposed by [WWWK95] for instance. The resulting performance in the second case however is generally poor.

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

Figure 1. Different Communication Models

2.2 Subscription Styles

When subscribing, a party expresses its interests in receiving certain messages. Rarely a subscriber is eager to receive all produced messages. Dividing the message space provides a means of confining the subscribers requirements. There are different approaches to the classification of messages. This can be done in advance by introducing a global classification (*topic-based*), or more dynamically by taking into consideration the nature of the messages that will be created during the lifetime of the system (*content-based*).

2.2.1 Topic-Based Publish/Subscribe

The classic publish/subscribe interaction model is based on the notion of *topics* or *subjects*, which basically resemble groups [Pow96]. Subscribing to a topic T can be viewed as becoming member of a group T. The topic abstraction however differs from the group abstraction by its more dynamic nature. While groups are usually disjoint sets of members (e.g., group communication for replication [Bir93]), 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 [TIB99]. 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).

2.2.2 Content-Based Publish/Subscribe

A next step to loosen the restrictions of communication models has been taken by the introduction of *content-based* publish/subscribe [CRW98, SA97]. This new feature gives even more flexibility to the application, by removing entirely the limitations of concretely defined distinct topics. Subscribers can announce their individual interests by specifying the properties of the event notifications they are interested in. The notifications or messages are therefore not

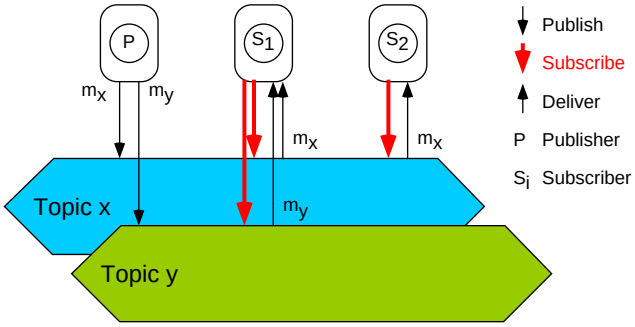

Figure 2. Topic-Based Subscribing

classified according to *arbitrarily* fixed criteria, but by their runtime properties. Each subscriber hence only receives the notifications that match entirely its individual criteria. The variations of these criteria have also to be taken into account. Thanks to the expressiveness of the content-based approach the dissemination of unrequired messages can be avoided.

Figure 3 shows the difference to topic-based subscribing. As a matter of fact, one can picture the message space as a single topic. Every subscriber announces its individual criteria on the messages. In the situation outlined in the figure, message m_1 contains ○ and therefore matches the criteria of S_2 (which is interested in messages containing ● or ○) and S_1 (only interested in ○). Message m_1 is thus delivered to both. The content of message m_2 though only matches the requirements of subscriber S_2 and is hence not delivered to subscriber S_1.

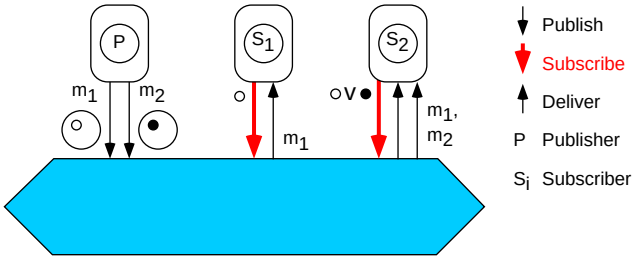

Figure 3. Content-Based Subscribing

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 two 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 publisher 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 [Ske98]. This type of interaction is called *pull model*.[5]

Although in general the push model seems more appropriate, certain applications may not be interested in receiving information as soon as possible, but only at precise moments. In those situations, a pull-style interaction 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 systems provide different *qualities of service*, in order to meet the demands of a variety of application purposes [AEM99, TIB99].[6] 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.

[5] *Blocking* is a more synchronous pull-type interaction, where a participant which tries to pull information is blocked until a new notification is available. Just like the request-reply model however, this variant lacks time decoupling.

[6] [TIB99] adopts the notion of *delivery service*.

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* [TIB99].

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 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 [Obj99] for instance, by being essentially distributed[7] and asynchronous. 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 [IBM95] and Java [JCF99] 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 [SL95]) as additional libraries. Figure 4 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* [BGL98] (*future type message passing* [YSTH87]), 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* [Car93]), or ignore it. Figure 5 compares the two paradigms. When programming with DACs, the 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, whereas with DACs, every time an information which is interesting for the registered party is created, it will be sent to it.

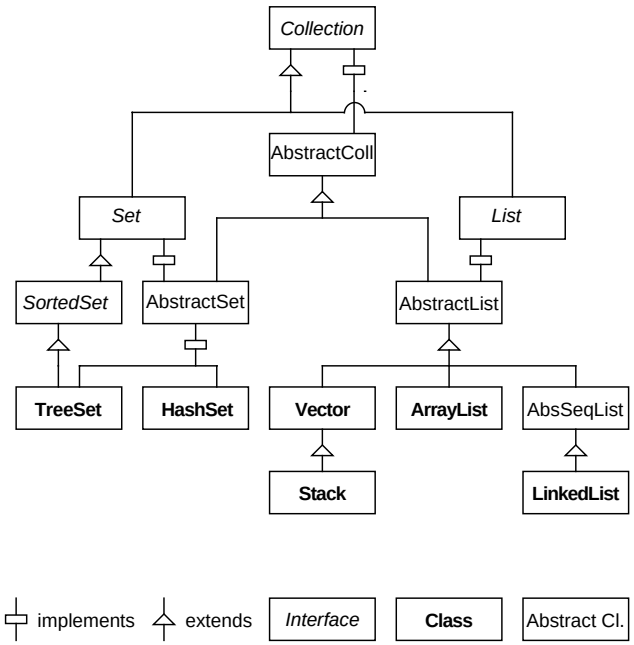

Figure 4. Collections in Java (excerpt)

[7] The distributed collections presented in [Obj99] are centralized collections that can be remotely accessed through RMI.

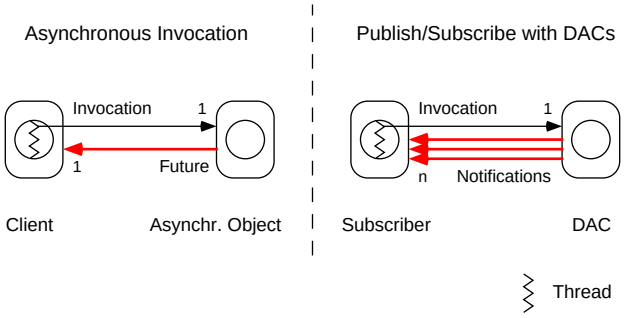

Figure 5. DACs vs. Future

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 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 6 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 7. It encapsulates the application message.

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

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

Figure 6. "Traditional" Topic-Based Publish/Subscribe

3.3.1 Content-Based Publish/Subscribe with DACs

We have explained how topic-based publish/subscribe can be realized through DACs in an object-oriented setting by considering event notifications as objects. We strive to go a step further by realizing content-based publish/subscribe which is more flexible by allowing a subscriber to express its individual requirements instead of conforming to a rigid classification of events. Moreover, DACs do not only provide *both* topic-based and content-based styles, but offer the possibility to combine the two publish/subscribe styles. Indeed, with the ability to apply a content-based subscription pattern to a set of topics, we get the best of both worlds.

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 topicName = "/Chat/Insomnia"`
Evaluation	`p.getTopic().equals("/Chat/Insomnia")`
Deliver	`m = p.getMsg()`

Figure 7. Topic-Based Publish/Subscribe with DACs

Existing content-based publish/suscribe systems allow the application to express its individual subscription based on the *contents* or *attributes* of the messages. In such systems [CRW99, SA97, SBCea98] which are mostly realized in procedural languages, a message can indeed best be pictured as a *record* with several *fields*. It seems problematic to apply such an approach without accessing the objects attributes and thus violating encapsulation. Conventional content-based publish/subscribe systems furthermore introduce a specialized query language to express conditions on the messages. These languages do not only expose the messages structure and content but also render the use of the system more difficult. Furthermore, they often only endulge a fixed set of patterns. The subscription argument given in Figure 8 would be passed directly as string to the publish/subscribe engine.

Message m	`public class ChatMsg {`
	`  public String sender;`
	`  ...`
	`}`

Criteria	sender of m is "Tom"
Argument	`String criteria = "sender is Tom"`
Evaluation	`m.sender.equals("Tom")`
Deliver	`m`

Figure 8. "Traditional" Content-Based Publish/Subscribe

Our approach removes these deficiencies by using *reflection* [KdRB91]. Reflection allows to represent and manip-

ulate characteristics of the language by the language itself. In particular, this allows to reflect properties of objects, in our case messages used to notify events, through the language itself. In fact, instead of expressing conditions on an object through its attributes, we query the object through its methods. Roughly spoken, our approach to specifying constraints is based on the association of a method and a result as seen in Figure 9. The method `getSender()` is evaluated on a message object m and the result is compared to the required value. The reflection properties of an object-oriented language give us the mechanisms to represent such constraints in terms of objects. Section 4 depicts more in detail how DACs allow to express and manipulate conditions as objects, instead of using a separate subscription language.

Message m	`public class ChatMsg {`
	`public String getSender(){...}`
	`...`
	`}`

Criteria	method `getSender()` of m returns "Tom"

Arguments	`Method sender = ...`
	(reference to method `getSender()`)
	`Object[] args = null`
	(arguments for the call, here empty)
	`Object res = "Tom"`
	(required result)

Evaluation	`m.getSender().equals("Tom")` i.e., `sender.invoke(m,args).equals(res)`
Deliver	`m`

Figure 9. Content-Based Publish/Subscribe with DACs

3.3.2 Type-Based Publish/Subscribe

In classical publish/subscribe systems, all event notifications are carried by messages of the same type (Figure 6). The topic is an attribute of that type which is introduced to regroup messages with a common meaning. When considering messages as objects however, these already express an affiliation through their type. In other terms, the notion of *event kind* is simply matched with that of an *event type*. That is, we use the type scheme of an "ordinary" programming language without explicitly introducing a topic hierarchy, a predicate-matcher tool or a specific notion of kind. Topics are represented by distributed asynchronous collections for specific types, and subscribing to a specific collection of which events are of a given type `InsomniaMsg` (Figure 10), implicitly means that the events of interest are those of type `InsomniaMsg`. This approach allows furthermore to use type polymorphism by generating automatic subscriptions to collections of subtypes. If a consumer subscribes to a collection of a type `ChatMsg` then a subscription can be automatically generated for any collection of which events are of a subtype of `ChatMsg` (e.g., `InsomniaMsg`).

Message m	`public class ChatMsg {...}`
	`public class InsomniaMsg`
	`extends ChatMsg {...}`

Criteria	m is of type `InsomniaMsg`
Argument	`Class imClass =` `Class.forName("InsomniaMsg")`
Evaluation	`m instanceof InsomniaMsg` i.e., `imClass.isInstance(m)`
Deliver	`m`

Figure 10. Type-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. Each of the three parts of this section describes one of the subscription styles we offer, namely *topic-based*, *content-based* and *type-based*. In the context of the second one, we introduce *Conditions* and *Accessors* as a means of representing constraints.

4.1 Topic-Based Publish/Subscribe

In our system, each *topic* is represented by a DAC, and is denoted by a name, like "Chat". 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. Existing publish/subscribe frameworks introduce specialized message types, e.g., [HBS98]. 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.[8]

Figure 11 summarizes the main methods of the base

[8] In order to be conveyable, a Java object should furthermore implement the `java.io.Serializable` interface [JLA99], which contains no methods.

DAC interface. The complete interface is given in Appendix A.1. 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 not denatured, and we connote them as *synchronous* in contrast to the methods added to express the *asynchronous* nature of publish/subscribe interaction specific to DACs. 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.

Synchronous Methods:

- `get()`. Similarly to a centralized collection, calling this method allows to retrieve objects. 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 [OMG98], this is called a *pullsupplier*.

The following *asynchronous* methods have been added to express the distributed asynchronous flavor of DACs.

Asynchronous Methods:

- `contains(Subscriber S,...)`. The effect, for instance, of invoking one of these two methods 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 `Subscriber` interface (Figure 12), through which it will be notified of events. There are different signatures for this method. The first variant given in Figure 11 can be used by a participant to subscribe to the topic represented by the DAC, whereas the second proposed signature allows a fine-grained specification of constraints for content-based filtering. This is covered in more detail in the next section.

- `containsAll(Subscriber S,...)`. These methods offer the same signatures than the two previous methods in Figure 11. The difference is that a subscription is generated for all subtopics of the topic represented by this DAC.

- `remove(Subscriber S,...)`. 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* [TIB99] in contrast to *one-for-each*,[9] implemented by the asynchronous `contains()` methods, where a message is sent to all. The same signatures can be found than for the asynchronous `contains()` and `containsAll()`.

- `clear(Subscriber S)`. While the conventional argument-less `clear()` method allows to erase all elements from the collection, this asynchronous variant expresses the action of *unsubscribing*.

4.2 Content-Based Publish/Subscribe

Existing content-based publish/subscribe implementations usually rely on a separate query language for a flexible expression of the constraints. This brings the burden of learning the language, and furthermore often only allows a limited set of patterns. Our approach uses Java language reflection properties to describe constraints. Java offers classes representing methods (`java.reflection.Method`) as well as classes themselves (`java.lang.Class`) and other specific Java language constructs.[10] These give a means to *describe an object by objects*. The idea is to use these reflection properties, and to offer a syntaxically related interface for an easier use.

The main constraints that can be defined on a Java object have been identified and sorted. For the sake of brevity we present only the most important ones:

[9] By using the formalism of [RW97], 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.

[10] There is also type `java.reflection.Field` representing an attribute. Dealing with such objects however means abandoning encapsulation.

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

{
  public Object get();
  public boolean contains(Object message);
  public boolean add(Object message);
  ...
  public boolean contains(Subscriber S);
  public boolean contains(Subscriber S,
                          Condition c);
  public boolean containsAll(Subscriber S);
  public boolean containsAll(Subscriber S,
                             Condition c);
  ...
  public boolean remove(Subscriber S);
  ...
  public void clear(Subscriber S);
  ...
}
```

Figure 11. Interface DAC (Excerpt)

```
public interface Subscriber {

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

Figure 12. Interface Subscriber

I. **Is object O of class C?**
This reflects the method `isInstance()` in the Java class `java.lang.Class`. That method is itself the dynamic counterpart to the language `instanceof` operator. One can expect the result to be *true* or *false*. When using type-based subscribing, this condition applies implicitly.

II. **Is object O equal to object O'?**
Every object can be compared to another object by using the `equals()` method, which is defined on each object. Again the result can be *true* or *false*.

III. **How does object O compare to object O'?**
This is only for objects implementing the `java.lang.Comparable` interface. Every object implementing that interface provides a method called `compareTo()`. The return value is of integer type, which indicates the order of the object compared to the second object. This presupposes that the objects manifest a natural ordering e.g., class `java.lang.Integer`.

By applying one of these tests to the message objects for filtering, only a coarse granularity can be achieved. The two following extensions improve the expressiveness considerably:

IV. The same tests can be applied also to any **object O'' which is a return value of a method of object O.**
That way, we can apply all the above constraints to *nested* method calls.

V. A **conjunction of any of these constraints** is of course more realistic than just a single one.
Therefore any logical combination of several conditions should be possible. These are called *filters* in [CRW98].

```
public interface Condition {

  public boolean add(Object msg,
     String topicName);
}
```

Figure 13. Condition Interface

4.2.1 Conditions

We use objects to express the above mentioned constraints. By creating an instance of class `Equals` (Figure 14, first constructor) and giving it a reference to an object O, we can express for instance that we are interested in all objects which are equal to O in the sense of the `equals()` method which is inherent in Java. Such condition objects are similar to the *argument filters* defined in [CRW98] and can be seen as unary predicate objects. They implement the `Condition` interface outlined in Figure 13. A message will be delivered to a subscriber iff the method `add()` of the condition object associated to its subscription returns *true* for that message object. [Obj99] describes a similar approach to predicate objects in Java. Our approach however focuses mainly on unary predicates.[11] Moreover, our efforts have been concentrated on finding a simple and intuitive way for the application programmer to create and combine our basic conditions. The application defining its constraints does not apprehend these as predicates, but just expresses its conditions for the message objects it is interested in. Therefore, we refer to these constructs as *conditions* rather than predicates.

There are thus two possibilities to express constraints describing the desired notifications:

- Create and combine instances of the condition classes provided in package `DACE.Conditions`. This minimizes the effort for the application programmer, since

[11] The second argument, representing the name of the topic is only used in certain cases, and will therefore be ignored henceforth.

we provide classes covering a large spectrum of applications. These classes furthermore provide hooks for a faster evaluation.

- Generate specific condition classes. This allows the application developer to generate custom tailored conditions. By implementing the `Condition` interface, the developer has full control of his criteria, and these can vary throughout the validity of the subscription without any further call to the DAC. [12]

```
public final class Equals
  implements Condition

{
  public Equals(Object to, boolean result)
  public Equals(Accessor obj, Object to,
                boolean result)
  public Equals(String name, Object to,
                boolean result)
  public Equals(String name, Object[][] params,
                Object to, boolean result)
  ...
  public boolean add(Object msg, String topicName)
}
```

Figure 14. Equals Class (Excerpt)

Simple subscription patterns involving only one condition are rather rare. We also provide operators to combine conditions. These can be seen as binary predicates, if the two conditions that will be combined are seen as the arguments. From our point of view, those two conditions are not the arguments. The argument of an operator is the message object for which it is evaluated at runtime. These operators hence implement the `Condition` interface as well.

Figure 15 shows the evaluation of a simple predicate representing an equality condition in the sense of the Java `equals()` method supported by all Java objects. When the condition object is created, an object to is passed as argument to it. When the predicate is evaluated for a message msg (which can be of any class), the `equals()` method is called on the message with to as argument. If the method call returns *true*, the message is delivered to the subscriber associated to the condition.

4.2.2 Accessors

Rather than expressing conditions on merely the message object itself, it seems more natural to put restrictions only on part(s) of those objects. This can be done by specifying restrictions on an object returned by a method of the message object. This possibility is given by using *accessor objects*. Such an object implements the `Accessor` interface shown in Figure 16, and basically represents a means to access a partial information on the runtime message object.

[12] This holds only true as long as the original condition object is queried. By making the condition object serializable, it can be transferred to another host, allowing the messages to be filtered earlier. This helps reducing the number of unnecessarily conveyed messages.

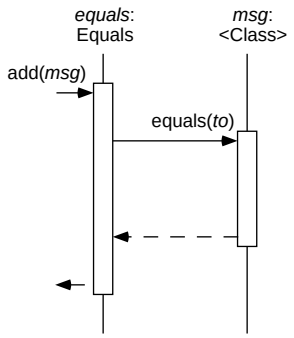

Figure 15. Predicate Evaluation

An accessor object can for instance represent an invocation of a given method with a given list of arguments. That method will be invoked on an incoming message passed to the accessor object by the encapsulating condition object. Figure 17 shows the class diagram for the `Equals` class. The second proposed constructor in Figure 14 allows to specify how to access part of the object through an accessor. In this case, the accessor object belongs to the generic class `Invoke` (classes `Equals` and `Invoke` are given in Appendix A.3 and Appendix A.4 respectively). An instance of that class bears a reference to a method, and can furthermore contain an array of objects representing the runtime arguments for that method. As explained previously, an instance of the `Method` class can *reify* any method of any class. This is made visible in the class diagram by putting the class name in $< ... >$.

The object returned by the accessor will be delivered if its comparison with the second argument that was passed to the `Equals` constructor is positive. In a nested way, it is possible to consider only the return value of a method of the object returned by the first accessor. In fact, when constructing an accessor object, it is possible to indicate a nested accessor object (this is shown in Figure 17 by the reference *nested*). The method referenced by the encapsulating accessor will be invoked on the returned value of the method of the first accessor. This offers an arbitrary fine granularity for the expression of conditions. The third variant shows a shortcut for the same purpose. The first argument denotes the name[13] of the access method(s) to be called (in a nested way). The accessors are created implicitly.

[13] In Java, obtaining a reference to a `Method` or a `Field` object always requires at least its name as it appears in the interface or class.

```
public interface Accessor {

  public Object get(Object msg);
}
```

Figure 16. Accessor Interface

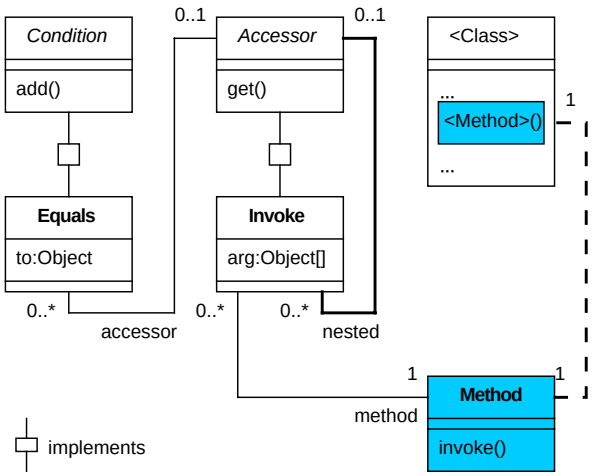

Figure 17. Class Diagram

Figure 18 illustrates the runtime interaction based on the previous example. When an event notification msg belonging to class $< Class >$ is produced, it is matched against the criteria of every subscriber through their respective condition objects (1). If an accessor was specified along with a method, evaluating the accessor (2) will result in invoking the method (3) object which triggers the effective method call on the message object (4). If the method object represents a method of the message objects class (if $< Method >$ is a method of the message objects class), the return value will be compared to the reference object by a call to the `equals()` method (5). Appendix B.1 shows more details in a concrete example.

4.3 Typed DACs

As mentioned earlier, our DACs support both topic-based and content-based publish/subscribe or a combination of both. Another feature we offer consists in providing a stronger typed interaction model. In fact, as explained in Section 4, type-based publish/subscribe can be seen as the projection of the name space to a type space, and as a consequence a DAC is bound to a single type. The DAC accepts only events of that type, which impacts the DAC interface. Thanks to a pre-compiler, type-specific DACs can be created automatically. During pre-compilation, a typed `DAC` interface is generated as well as typed `Subscriber` and `Condition` interfaces.

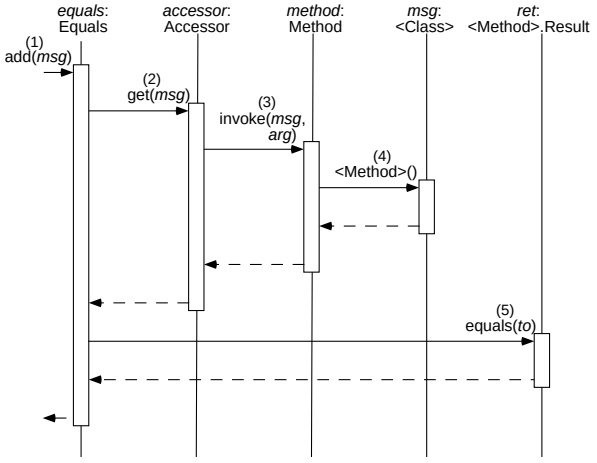

Figure 18. Accessor Interaction

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 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 19.

5.2 Duplicates

Just like it is possible to have duplicate elements in centralized collections, it is possible in Distributed Asyn-

chronous Collections that a same message is delivered more than once. 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 19. 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.

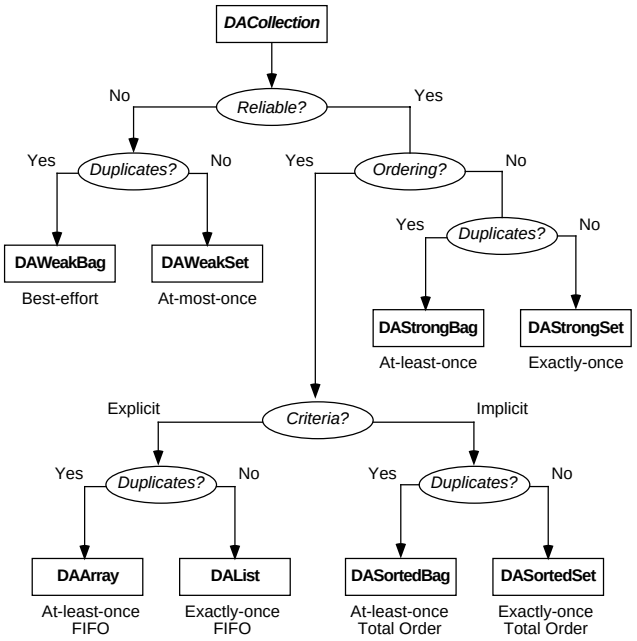

Figure 19. DAC Framework

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 19 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 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.

6.1 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 20 shows the different layers in our implementation. These layers do not necessarily correspond to Java classes, but represent protocol layers.

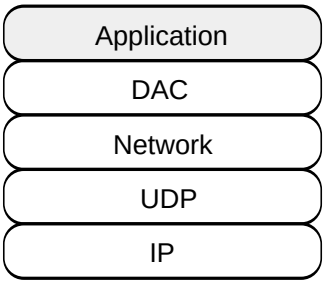

Figure 20. Layers

- **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.

6.2 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 4 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.

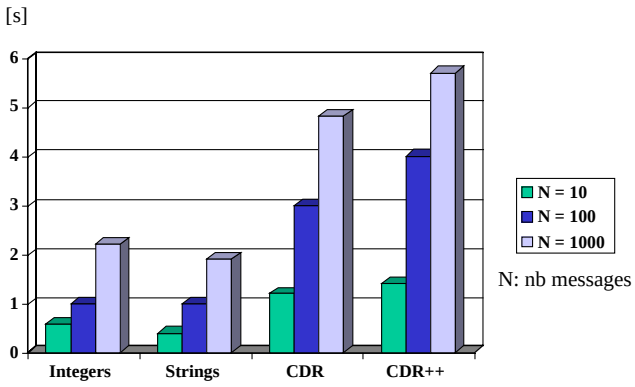

Figure 21. Latency

Figure 21 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.

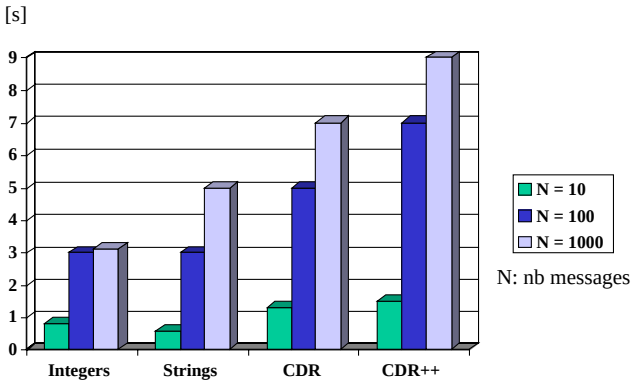

Figure 22. Throughput

Figure 22 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:

- 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.

7 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.

7.1 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 [OMG98] and Sun [HBS98, AOS+99, Col99].

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* [HBS98] 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.

The *Java Distributed Event Specification* [AOS+99] 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* [FHA99] to provide support for topic-based publish/subscribe notification. Inspired by Linda [Gel85], 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* [Col99] 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 lacks content-based publish/subscribe, and the Java Messaging Service introduces an explicit message class and uses Java `Properties` to describe constraints in the case of content-based subscribing. This leads to defining a subscription language, while oppositely DACs exploit the reflection facilities of the language.

7.2 Topic-Based Systems

Most industrial strength solutions involve topic-based publish/subscribe, but offer less support for content-based subscription facilities. *Smartsockets* [Cor99] or *TIB/Rendezvous* [TIB99] 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 offer API's in object-oriented languages like Java. These solutions however did not undergo a fundamentally object-oriented design. In addition, they mainly offer the rather rigid topic-based subscription style.

7.3 Content-Based Systems

Most approaches to content-based publish/subscribe are outcome of academic research, like *Siena* [CRW98] or *Elvin* [SA97]. But also industrial players have brought up interesting solutions in the context of content-based publish/subscribe, as shown by *Gryphon* [BCM+99].

Gryphon takes a new direction by introducing information flow graphs, which describe the routing and manipulation of streams of events from information providers to information consumers. Reflection is also present in Gryphon, but with another objective than to express content-based subscribing. In fact, protocol messages, concerning for instance subscriptions, are handled just like notifications created by applications. Much effort has also been made to optimize the matching algorithms.

Siena's strength is based on its scalability, since it was designed especially for wide area networks. Siena also explores different QoS, but more in combination with transactions and security. Filters are used to express content-based subscribing, and patterns allow to express the combination of simple events.

Elvin covers mainly content-based publish/subscribe. Requirements are expressed through a specially developed subscription grammar. For performance reasons Elvin was implemented in C, and notifications are therefore records, like in most classical content-based systems.

These approaches have driven the evolution of content-based publish/subscribe. In contrast to our DACs though, they use specific query languages to express requirements, and do furthermore not integrate several subscription styles. Some of the ideas developed in these projects are very interesting and have motivated us to adapt them to an object-oriented setting [SBS98].

7.4 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, e.g., for Java. *JGL* [Obj99] and the `util.concurrent` [Lea99] package offer more elaborate collection types.

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. JGL also offers algorithms to process collections, including predicate and func-

tion objects. As depicted in Section 4.2 however, they concentrate more on the inside view of a predicate, and less on the ability of creating and combining them easily in order to intuitively express a more complex constraint. Synchronization is an issue we do not address with our DACs, but could be the topic of future work.

8 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 [WWWK94, Lea97, Gue99] 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 [BGL98], 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. Furthermore, existing approaches to content-based publish/subscribe usually introduce new languages to express filters. DACs are lightweight publish/subscribe abstractions: they can be introduced through a library approach and they exploit the reflection facilities of the language to express content-based filtering, removing the need for an additional query language.

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 presented in the appendix. 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 specific algorithms to realize efficient matching, specially in a mobile environment, where nodes might be disconnected, and objects might migrate from a node to another [JLHB88].

References

[AEM99] M. Altherr, 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.

[AOS+99] K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, and J. Wollrath. *The Jini Specification*. Addison Wesley, June 1999.

[BCM+99] G. Banavar, T. Chandra, B. Muhkerjes, J. Nagarajarao, 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)*, 1999.

[BGL98] J.P. Briot, R. Guerraoui, and K.P. Löhr. Concurrency, distribution and parallelism in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[Bir93] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[Car93] D. Caromel. Towards a method of object-oriented concurrent programming. In *Communications of the ACM*, volume 36, pages 90–102, September 1993.

[Col99] M. Colan. InfoBus 1.2 specification. Technical report, Sun Microsystems Inc., February 1999.

[Cor99] Talarian Corporation. *SmartSockets White Paper*. http://www.talarian.com/products/, 1999.

[CRW98] A. Carzaniga, 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, http://www.cs.colorado.edu/ carzanig/papers/, August 1998.

[CRW99] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Engineering Distributed Objects '99*, http://www.cs.colorado.edu/ carzanig/papers/, May 1999.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison Wesley, June 1999.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, January 1985.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gue99] R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. *Informatik*, 2, April 1999.

[HBS98] M. Happner, R. Burridge, and R. Sharma. Java Message Service. Technical report, Sun Microsystems Inc., October 1998.

[HJ99] 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.

[IBM95] IBM. *Smalltalk Tutorial*. http://www.smalltalksystems.com/references.htm, 1995.

[JCF99] *The Java Collections Framework*. http://java.sun.com/products/jdk/1.2/docs/, 1999.

[JLA99] *The Java Platform 1.2 API Specification*. http://java.sun.com/products/jdk/1.2/docs/api/, 1999.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133, February 1988.

[KdRB91] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Koe99] P. Koenig. Messages vs. objects for application integration. *Distributed Computing*, 2(3):44–45, April 1999.

[Lea97] D. Lea. Design for open systems in Java. In *Second International Conference on Coordination Models and Languages*, http://gee.cs.oswego.edu/dl/coord/, 1997.

[Lea99] D. Lea. *Overview of package util.concurrent Release 1.2.5*. http://gee.cs.oswego.edu/dl/classes/, October 1999.

[Obj99] ObjectSpace. *JGL - Generic Collection Library*. http://www.objectspace.com/products/jgl/, 1999.

[OMG98] OMG. *CORBAservices: Common Object Services Specification*. OMG, December 1998.

[OPSS93] B. Oki, M. Pfluegl, 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.

[Pow96] D. Powell. Group communications. *Communications of the ACM*, 39(4):50–97, April 1996.

[RW97] 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 Soft ware Engineering*, September 1997.

[SA97] 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 (AUUG97)*, http://www.dtsc.edu.au/, September 1997.

[SBCea98] R. Strom, G. Banavar, T. Chandra, and M. Kaplan et al. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE '98)*., November 1998.

[SBS98] D.C. Sturman, G. Banavar, and R. Strom. Reflection in the Gryphon message brokering system. In *Reflection Workshop of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998.

[Ske98] D. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*. http://www.vitria.com, 1998.

[SL95] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Silicon Graphics Inc., October 1995.

[SV97] D. Schmidt and S. Vinoski. Overcoming drawbacks in the OMG Event Service. *SIGS C++ Report magazine*, 10, June 1997.

[TIB99] TIBCO. *TIB/Rendezvous White Paper*. http://www.rv.tibco.com/whitepaper.html, 1999.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Inc., November 1994.

[WWWK95] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. Events in an RPC based distributed system. Technical report, Sun Microsystems Laboratories Inc., November 1995.

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. *Object-Oriented Concurrent Programming*, chapter Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, pages 55–89. MIT Press, 1987.

Appendix A DAC Interfaces and Classes

A.1 DAC Interface

We present here the core interface, namely the `DAC` interface, of which interfaces for more specialized Distributed Asynchronous Collections inherit.

```
1   package DACE;
2
3   public interface DAC extends java.util.Collection {
4
```

As explained in Section 3.3, inserting an object into a DAC comes to publishing that object. When calling method `add()`, the object passed as argument is sent to all subscribers.

```
5     /** Publishing */
6
7     /**
8      *  Publish a message for this DAC.
9      *  @param message The message
10     *  @return True if the message was successfully published.
11     *  @see java.util.Collection
12     */
13    public boolean add(Object message);
14
```

The *pull* model is implemented by actively asking the DAC for a new object. If no new message has been received yet, the `null` value is returned.

```
15    /** Pulling */
16
17    /**
18     *  Pull a new message
19     *  @return An object, if there's one, null otherwise.
20     */
21    public Object get();
22
```

When subscribing with *one-for-each* semantics (a same message is delivered to *each* subscriber), one of the following methods must be used for pure topic-based subscribing. The `containsAll()` variant on line 39 generates subscriptions for all subtopics as well.

```
23    /** Topic-based subscribing (one-for-each) */
24
25    /**
26     *  Subscribe to this DAC.
27     *  @param s The subscriber callback object
28     *  @return True if the subscription was possible.
29     *  @see Subscriber
30     */
31    public boolean contains(Subscriber s);
32
33   /**
34     *  Subscribe to this DAC with subtopics.
35     *  @param s The subscriber callback object
36     *  @return True if the subscription was possible.
37     *  @see Subscriber
38     */
39    public boolean containsAll(Subscriber s);
40
```

With *one-for-each* semantics, the next two methods offer the possibility to specify finer constraints by combining with a content-based condition pattern. Again, the `containsAll()` method applies the same subscription pattern to all subtopics.

```
41    /** Content-based subscribing (one-for-each) */
42
43    /**
44     *  Subscribe to this DAC with condition(s).
45     *  @param s The subscriber callback object
46     *  @param c List of constraints
47     *  @return True if the subscription was possible.
48     *  @see Subscriber
49     *  @see Conditions
50     */
51    public boolean contains(Subscriber s, Condition c);
52
53    /**
54     *  Subscribe to this DAC with its subtopics with condition(s).
55     *  @param s The subscriber callback object
56     *  @param c List of constraints
57     *  @return True if the subscription was possible.
58     *  @see Subscriber
59     *  @see Conditions
60     */
61    public boolean containsAll(Subscriber s, Condition c);
62
```

In contrast to the `contains()` and `containsAll()` methods, the `remove()` and `removeAll()` methods allow to express *one-for-all* semantics, which means that a message is delivered to one within all subscribers. These two first methods provide for pure topic-based subscribing.

```
63    /** Topic-based subscribing (one-for-all) */
64
65    /**
66     *  Subscribe to this DAC.
67     *  @param s The subscriber callback object
68     *  @return True if the subscription was possible.
69     *  @see Subscriber
70     */
71    public boolean remove(Subscriber s);
72
73    /**
74     *  Subscribe to this DAC with subtopics.
75     *  @param s The subscriber callback object
76     *  @return True if the subscription was possible.
77     *  @see Subscriber
78     */
79    public boolean removeAll(Subscriber s);
80
```

Again with *one-for-all* semantics, these two variants support content-based subscribing. Just like the previous `removeAll()` method (line 79), the one on line 101 triggers subscriptions to all subtopics.

```
81    /** Content-based subscribing (one-for-all) */
82
83    /**
84     *  Subscribe to this DAC with condition(s).
85     *  @param s The subscriber callback object
86     *  @param c List of constraints
87     *  @return True if the subscription was possible.
88     *  @see Subscriber
89     *  @see Conditions
90     */
91    public boolean remove(Subscriber s, Condition c);
92
```

```
93    /**
94     *  Subscribe to this DAC with subtopics with condition(s).
95     *  @param s The subscriber callback object
96     *  @param c List of constraints
97     *  @return True if the subscription was possible.
98     *  @see Subscriber
99     *  @see Conditions
100    */
101   public boolean removeAll(Subscriber s, Condition c);
102
```

A subscriber can be unsubscribed through this method.

```
103   /** Unsubscribing */
104
105   /**
106    *  Unsubscribe this subscriber from this DAC.
107    *  @param s The subscriber to be removed.
108    */
109   public void clear(Subscriber s);
110
```

The next methods give access to the identity of the DAC. The first method returns the name of the topic which this DAC represents. The second method must be implemented by every Java object. In the case of a DAC, it returns *true* if the object it is compared to is a DAC representing the same topic. Finally, the last method (line 130) returns a unique hash code for this DAC.

```
111   /** Identity */
112
113   /**
114    *  Returns the name of the topic represented by this DAC.
115    *  @return The name of the topic.
116    */
117   public String getName();
118
119   /**
120    *  Compares the specified object with this dac for equality.
121    *  @param dac The DAC containing the topic to compare to
122    *  @return True if the object represents the same topic.
123    *  @see Object
124    */
125   public boolean equals(Object dac);
126
127   /**
128    *  Returns the hash code value for this object.
129    */
130   public int hashCode();
131
```

The remaining methods are all inherited from the original `java.util.Collection` interface. Their meanings have been adapted. The first method allows to query whether an object was published under this topic. The second method allows to verify whether a DAC represents a subtopic of the topic represented by this DAC.

```
132   /** Relationship */
133
134   /**
135    *  Verify whether a message belongs to this DAC.
136    *  @param message The message
137    *  @return True if the message belongs to this DAC, false otherwise.
138    *  @see java.util.Collection
139    */
140   public boolean contains(Object message);
141
```

```
142    /**
143     *  Verify whether a DAC represents a specialization of this DAC.
144     *  @param dac The DAC representing the other topic
145     *  @return True if the topic is a specialization of this topic, false otherwise.
146     *  @see java.util.Collection
147     */
148    public boolean containsAll(java.util.Collection dac);
149
```

Methods `isEmpty()` and `size()` give information about the number of messages that have not been consumed yet. They are particularly useful in combination with the *pull* model. Method `clear()` allows to purge the message buffer.

```
150    /** Size */
151
152    /**
153     *  Returns true if this DAC contains no messages.
154     *  This indicates if any messages have not been consumed yet.
155     */
156    public boolean isEmpty();
157
158    /**
159     *  Returns the number of elements in this collection.
160     *  This indicates the number of messages that have not been consumed yet.
161     */
162    public int size();
163
164    /**
165     *  Clear this DAC.
166     *  @see java.util.Collection
167     */
168    public void clear();
169
```

These last standard methods are used to browse the message buffer backing this DAC instance. In combination with the `pull` model they allow to retrieve several values at a time.

```
170    /** Buffer */
171
172    /**
173     *  Returns an iterator over the messages for this DAC.
174     *  @return An iterator
175     *  @see java.util.Collection
176     *  @see java.util.Iterator
177     */
178    public java.util.Iterator iterator();
179
180    /**
181     *  Returns an array containing all the messages.
182     *  @return An array
183     *  @see java.util.Collection
184     */
185    public Object[] toArray();
186
187    /**
188     *  Returns an array containing all of the messages in this DAC
189     *  whose runtime type is that of the specified array.
190     *  @param a An array with the given runtime type
191     *  @return An array of same type
192     *  @see java.util.Collection
193     */
194    public Object[] toArray(Object[] a);
195
196 }
```

A.2 Interfaces for Subscribing

Below are further interfaces used for subscribing. Interface `Subscriber` is used for the callbacks from the DAC. The `contains()` method is called by the DAC whenever a message corresponding to the subscription criteria is received.

```
1   package DACE;
2
3   public interface Subscriber {
4
5     /**
6      *  Callback method for delivery of messages
7      *  @param message The incoming message
8      *  @param topicName The precise associated topic name
9      */
10    public void contains(Object message,
11                         String topicName);
12  }
```

Interface `Condition` is used to express subscription patterns. An object implementing that interface must be passed to the DAC upon subscription in order to benefit from content-based publish/subscribe (e.g., by a call to method `contains()` on line 51, Appendix A.1). The condition object will be querried to evaluate whether a message should be delivered to the associated subscriber. Appendix A.3 shows an implementation example.

```
1   package DACE;
2
3   public interface Condition {
4
5     /**
6      *  Query whether the message should be delivered
7      *  @param message The message that must be matched
8      *  @param topicName The precise name of the topic
9      *  @return True if the message should be delivered
10     */
11    public boolean add(Object message,
12                       String topicName);
13  }
```

The `Accessor` interface is used in combination with the previous interface for content-based publish/subscribe. Through the `add()` method an accessor object gives access to a partial information on the object passed as argument. An example is given in Appendix A.4.

```
1   package DACE;
2
3   public interface Accessor {
4
5     /**
6      *  Get a component of the message.
7      *  @param message The message object
8      *  @param topicName The precise name of the topic
9      *  @return The component of the message object
10     */
11    public Object get(Object message,
12                      String topicName);
13  }
```

A.3 Equals Class

In the following we present the `Equals` condition class introduced in Section 4.2.

```
1   package DACE.Conditions;
2
3   import DACE.*;
4   import DACE.Conditions.*;
5   import DACE.Accessors.*;
6
7   public final class Equals implements Condition {
8
```

These are the arguments of this equality condition. The accessor object, denoted by `invoke`, is applied (if not `null`) to a message object, in order to compare the returned value to `to`. The result is matched against `requiredResult`.

```
9     private Accessor invoke = null;
10
11    private Object to = null;;
12
13    private boolean requiredResult = true;
14
```

This first constructor is used to express an equality requirement between runtime message objects and the object given as argument. The second argument indicates the needed result of the comparison.

```
15    /**
16     *  Indicate whether a message object is equal to a given object.
17     *  @param to Object the message object is compared to.
18     *  @param requiredResult What the result of the comparison should be
19     */
20    public Equals(Object to,
21                  boolean requiredResult) {
22
23        this.to = to;
24        this.requiredResult = requiredResult;
25      }
26
```

This constructor allows to put an equality restriction only on a return value of a method of the message objects. This variant accepts only methods without arguments (*pure* accessor) to be called. The method name is given as URL-like string (see Section B.1). Nested method calls can be expressed as well. Accessors are created implicitly: the accessor for the first method (that will be called on the message object) is created on line 48, while nested accessors are created and connected on line 50.

```
27    /**
28     *  Indicate whether a message object is equal to a given object.
29     *  If the object is the message itself: name = "";
30     *  If the object is returned by a method call: name = "<I>method-name</I>";
31     *  Recursively: name = "<I>method1-name/.../methodn-name</I>"
32     *  @param name Name of the argument-less method(s)
33     *  @param to Object the message object is compared to.
34     *  @param requiredResult What the result of the comparison should be
35     */
36    public Equals(String name,
37                  Object to,
38                  boolean requiredResult)
39      throws InitializationException {
40
41        this(to, requiredResult);
42        if (name == null)
43          throw new InitializationException("Missing name");
44        String[] names = Utils.getNames(name);
45        if (names.length != 0) {
46          Accessor nested = new Invoke(names[0], null);
47          for (int i = 1; i < names.length; i++)
48            nested = new Invoke(nested, names[i], null);
49          this.invoke = nested;
50        }
51      }
52
```

This variant differs from the previous one by allowing to specify methods with argument lists.

```
53    /**
54     *  Indicate whether a message object is equal to a given object.
55     *  The name of the method: name = "<I>method-name</I>";
56     *  Recursively: name = "<I>method1-name/.../methodn-name</I>"
57     *  @param name Name of the method(s)
58     *  @param params A list of arguments for each method call
59     *  @param to Object the message object is compared to.
60     *  @param requiredResult What the result of the comparison should be
61     */
62    public Equals(String name,
63                  Object[][] params,
64                  Object to,
65                  boolean requiredResult)
66      throws InitializationException
67
68      {
69        this(to, requiredResult);
70        if (name == null)
71          throw new InitializationException("Missing name");
72        String[] names = Utils.getNames(name);
73        if (names.length != params.length)
74          throw new InitializationException("Wrong number of argument lists");
75        if (names.length != 0) {
76          Accessor nested = new Invoke(names[0], params[0]);
77          for (int i = 1; i < names.length; i++)
78            nested = new Invoke(nested, names[i], params[i]);
79          this.invoke = nested;
80        }
81      }
82
```

This variant is based on explicit accessor creation. It offers the same functionality than the previous constructor. The main difference remains in the way a method is specified. The previous constructor uses the method name, while it is possible with accessors to specify methods as objects. Appendix A.4 elucidates the difference in more detail through a sample accessor.

```
83    /**
84     *  Indicate whether a message object is equal to a given object.
85     *  @param obj Accessor object that returns a component of the message
86     *  @param to Object the message object is compared to.
87     *  @param requiredResult What the result of the comparison should be
88     */
89    public Equals(Accessor obj,
90                  Object to,
91                  boolean requiredResult)
92      throws InitializationException
93
94      {
95        this(to, requiredResult);
96        this.invoke = obj;
97      }
98
```

These following methods give access to the attributes of the condition. They are needed by the `equals()` method, to compare two instances of this class.

```
99    /**
100    *  @return The required result for this condition.
101    *  @see equals
102    */
103   boolean getRequiredResult() { return requiredResult; }
104
```

```
105   /**
106    *  @return The accessor object for this condition.
107    *  @see equals
108    */
109   DACE.Accessor getAccessor() { return invoke; }
110
111   /**
112    *  @return The object to compare to.
113    *  @see equals
114    */
115   Object getTo() { return to; }
116
```

The `add()` method is called whenever this condition should be evaluated for a message. It returns *true* if the message satisfies the runtime condition expressed by this instance of `Equals`. If an accessor exists, it is evaluated first (the nested accessors, if any, are recursively invoked).

```
117   /**
118    *  Query whether the message should be delivered
119    *  @param message The message that must be matched
120    *  @param topicName The precise name of the topic
121    *  @return True if the message should be delivered
122    *  @see DACE.DAC
123    */
124   public boolean add(Object message,
125                      String topicName)
126
127     {
128       Object returnObj = null;
129       if (invoke == null)
130         return (message.equals(to) == requiredResult);
131       else
132         returnObj = (invoke.get(message, topicName));
133       if (returnObj == null)
134         return false;
135       else
136         return (returnObj.equals(to) == requiredResult);
137     }
138
```

This method compares two instances of this class for equality. When optimizing condition evaluation, this method permits to avoid the evaluation of redundant conditions.

```
139   /**
140    *  Indicate whether an object is equal to this Condition.
141    *  Overrides the standard implementation, makes a recursive test.
142    *  @param to Object this Condition is compared to.
143    *  @return Result of the comparison
144    */
145   public boolean equals(Object to)
146
147     {
148       try {
149         Equals other = (Equals)to;
150         return (requiredResult == other.getRequiredResult()
151                 && to.equals(other.getTo())
152                 && invoke.equals(other.getAccessor()));
153       } catch(java.lang.ClassCastException cce) {
154         /* Object is not of same class */
155         return false;
156       }
157     }
158
159 }
```

A.4 Invoke Class

Below we list the `Invoke` accessor class which allows to obtain partial information on a message through its methods in order to express conditions only on parts of the messages.

```
1   package DACE.Accessors;
2
3   import java.lang.reflect.*;
4   import java.util.*;
5   import DACE.*;
6
7   public final class Invoke implements Accessor
8
9   {
10
```

These are the arguments of this accessor. A method can either be represented by its name (`methodName`), or a method object (`method`). If this accessor represents a method call with arguments, `args` will be initialized. A possible nested method call is represented by a reference to the corresponding accessor (`nested`).

```
11    private String methodName = null;
12
13    private Method method = null;
14
15    private Object[] args = null;
16
17    private Accessor nested = null;
18
```

This first constructor allows to specify a method by its name, along with a list of arguments, which can be empty.

```
19    /**
20     *  Constructor for one method given by name.
21     *  @param methodName The name of the method.
22     *  @param args The arguments for the call. Can be null.
23     */
24    public Invoke(String methodName,
25                  Object[] args)
26      throws InitializationException
27
28      {
29        if (methodName == null || methodName.length() == 0)
30          throw new InitializationException("No method name specified");
31        this.methodName = methodName;
32        this.args = args;
33      }
34
```

To use this constructor, the application must use reflection explicitly.

```
35    /**
36     *  Constructor for one method given by reference.
37     *  Attention! The class of the method must be the class
38     *  of the runtime message.
39     *  @param method The method object.
40     *  @param args The arguments for the call. Can be null.
41     */
42    public Invoke(Method method,
43                  Object[] args)
44      throws InitializationException
45
46      {
47        if (method == null)
48          throw new InitializationException("Method object null");
49        this.method = method;
50        this.methodName = method.getName();
51        this.args = args;
52      }
53
```

Constructor for nested method calls. The corresponding method names are given by a URL-like string (first argument).

```
54    /**
55     *  Constructor for nested methods given by names.
56     *  @param methodNames The names of the methods.
57     *  @param args The arguments for the calls.
58     */
59    public Invoke(String methodNames,
60                  Object[][] args)
61      throws InitializationException
62
63      {
64        String[] names = Utils.getNames(methodNames);
65        if (names.length == 0)
66          throw new InitializationException("No methods specified");
67        if (args == null)
68          throw new InitializationException("Received null arguments");
69        if (names.length != args.length)
70          throw new InitializationException(
71            "Unequal number of methods and argument lists");
72        this.methodName = names[names.length - 1];
73        this.args = args[names.length - 1];
74        Object[] nestedArgs = new Object[names.length - 1];
75        String[] nestedNames = new String[names.length - 1];
76        System.arraycopy(args, 0, nestedArgs, 0, nestedArgs.length);
77        System.arraycopy(names, 0, nestedNames, 0, nestedNames.length);
78        nested = new Invoke(Utils.getURL(nestedNames), nestedArgs);
79      }
80
```

The next constructor gives the possibility to use nested accessors. The method associated to this accessor is specified by its name. Note that the nested accessor(s) can be specified by name *or* by using directly reflection.

```
81    /**
82     *  Constructor for nested methods. Add one by name.
83     *  @param methodName The name of the method.
84     *  @param args The arguments for the call.
85     */
86    public Invoke(Accessor nested,
87                  String methodName,
88                  Object[] args)
89      throws InitializationException
90
91      {
92        this(methodName, args);
93        this.nested = nested;
94      }
95
```

This constructor gives the possibility to use nested accessors as well, but the method represented by the top-level accessor is specified by a method object. Again, the nested accessor(s) can be specified by name or by using directly reflection.

```
96    /**
97     *  Constructor for nested methods. Add one by reference.
98     *  Attention! The class of the method must be the class
99     *  of the runtime message.
100    *  @param method The method object.
101    *  @param args The arguments for the call.
102    */
103   public Invoke(Accessor nested,
104                 Method method,
105                 Object[] args)
106     throws InitializationException
107
108     {
109       this(method, args);
110       this.nested = nested;
111     }
112
```

As explained in Section 4.2.2, an accessor object is evaluated on an object (message) through this method. The method associated to an accessor can be either given by its name or by a reference to the `Method` object. In the first case, a reference to the method object will be obtained at runtime in the beginning of the evaluation of this accessor (line 137). This requires that the runtime message object implements a method of that name with a signature corresponding to the arguments list. In the latter case, the method object, which is bound to a class, is already given. This expresses an implicit restriction on the type of the runtime message objects.

At line 125, the nested accessor (if any) is evaluated first. Lines 130 to 138 are necessary to get the method object with the signature corresponding to the argument list (if this accessor was initialized with a method *name*). At line 139, the method is finally invoked.

```
113   /**
114    *  Get a component of the message.
115    *  @param message The message object.
116    *  @param topicName The precise name of the topic.
117    *  @return The component of the message object.
118    */
119   public Object get(Object message,
120                     String topicName)
121
122     {
123       Object returnObject = null;
124       if (nested != null) {
125         Object msg = nested.get(message, topicName);
126         message = msg;
127       }
128       try {
129         if (method == null && message != null) {
130           Class[] argsClasses = null;
131           if (args != null && args.length != 0) {
132             argsClasses = new Class[args.length];
133             for (int i = 0; i < argsClasses.length; i++) {
134               argsClasses[i] = args[i].getClass();
135             }
136           }
137           method = message.getClass().getMethod(methodName, argsClasses);
138         }
139         returnObject = method.invoke(message, args);
140       } catch (NoSuchMethodException nsme) {
141         /* This method does not exist */
142       } catch (InvocationTargetException ite) {
143         /* Exception while invoking method */
144       } catch (IllegalAccessException iae) {
145         /* Class method may not be called */
146       } catch (IllegalArgumentException iae) {
147         /* Wrong signature for method call */
148       } catch (SecurityException se) {
149         /* Problem with SecurityManager */
150       } catch (Exception ex) {
151         /* Other problem */
152         ex.printStackTrace();
153       }
154       return returnObject;
155     }
156
```

These following methods give access to the attributes of the accessor. They are needed by the `equals()` method below, to compare two instances of this class.

```
157   /**
158    *  @return The method name.
159    *  @see equals
160    */
161   String getMethodName() { return methodName; }
162
```

```
163   /**
164    *  @return The arguments associated to the method.
165    *  @see equals
166    */
167   Object[] getArgs() { return args; }
168
169   /**
170    *  @return The nested accessor if any.
171    *  @see equals
172    */
173   Accessor getNested() { return nested; }
174
175   /**
176    *  @return The method object if any.
177    *  @see equals
178    */
179   Method getMethod() { return method; }
180
```

This method compares two instances of this class for equality. When optimizing condition and accessor evaluation, this method permits to avoid the evaluation of redundant accessors.

```
181   /**
182    *  Indicate whether an object is equal to this Accessor.
183    *  Overrides the standard implementation, makes a recursive test.
184    *  @param to Object this Accessor is compared to.
185    *  @return Result of the comparison
186    */
187   public boolean equals(Object to)
188
189     {
190       try {
191         Invoke other = (Invoke)to;
192         Object[] otherArgs = other.getArgs();
193         boolean argsEqual = true;
194         if (args == null || args.length == 0)
195           argsEqual = ( otherArgs == null || otherArgs.length == 0);
196         else if (args.length != otherArgs.length)
197           argsEqual = false;
198         else
199           for(int i = 0; i < args.length; i++)
200             if (!args[i].equals(otherArgs[i]))
201               argsEqual = false;
202         return (methodName.equals(other.getMethodName())
203                 && nested.equals(other.getNested())
204                 && argsEqual);
205       } catch(java.lang.ClassCastException cce) {
206         /* Object is not of same class */
207         return false;
208       }
209     }
210
211 }
```

Appendix B Putting DACs to Work

B.1 Programming with DACs

We describe here a simple example application using the flexibility of Distributed Asynchronous Collections. It shows how to implement *chat sessions* based on simple DACs. The complete code is given in Appendix B.2.

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 23). 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 24 shows class `ChatMsg`, which represents a possible message class for this application.

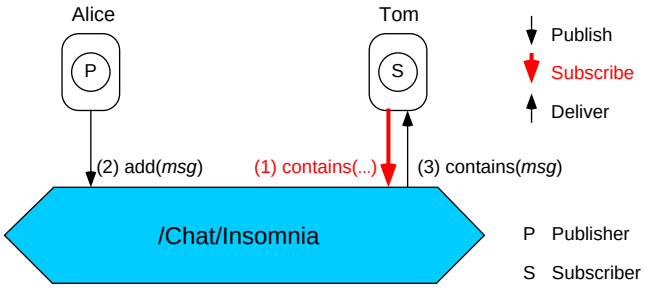

Figure 23. Chatters

```
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; }
}
```

Figure 24. Event Class for Chat Example

B.1.1 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 25 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 favourable 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 mychat =
  new DAStrongSet("/Chat/Insomnia");
String me = "Alice";
ChatMsg msg = new ChatMsg(me, "Hi everyone");
mychat.add(msg);
```

Figure 25. Publishing a Message

B.1.2 Topic-Based Subscribing

In order to subscribe to a topic an interested party must provide a callback object implementing the `Subscriber` interface (see Appendix A.2). 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". Our solution offers several ways to subscribe to a topic, specifying different arguments or constraints. Figure 23 shows the interactions with the DAC, and Figure 26 shows the corresponding code for a subscriber.

```
class ChatSubscriber
  implements Subscriber

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

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

Figure 26. Topic-Based Publish/Subscribe with DACs

```
DASet sleeplessChatters =
  new DAStrongSet("/Chat/Insomnia");
Condition onlyAlice =
  new Equals("/getSender", "Alice", true);
Subscriber sub = new ChatSubscriber();
sleeplessChatters.contains(sub, onlyAlice);
```

Figure 27. Content-Based Pub/Sub with DACs

B.1.3 Content-Based Subscribing

When choosing pure topic-based subscription mode, only a callback object is required. As mentioned in Section 3, DACs enable to combine for instance topic-based and content-based subscribing. This offers more flexibility than pure topic-based publish/subscribe, by allowing more dynamically defined constraints, which delineate the messages the applications are instantaneously interested in.

In order to benefit from this advanced feature, the application programmer must provide a condition object as revealed in Section 4. That object will evaluate for every received message, whether it matches the requirements or not. The condition object, to which the decision of delivery is delegated, can also be implemented by the application programmer. In this example however, we make use of a predefined condition object.

Figure 27 shows a simple example of content-based publish/subscribe. Suppose that Tom is only more interested in what Alice has to say, and defines a corresponding condition. This example uses implicit accessor creation, as explained in Section 4.2. Note the analogy between the string representing the topic, and the string representing the access to the message object. An even better illustration of the similarity between the two key types can be found on line 53 in Appendix B.2.

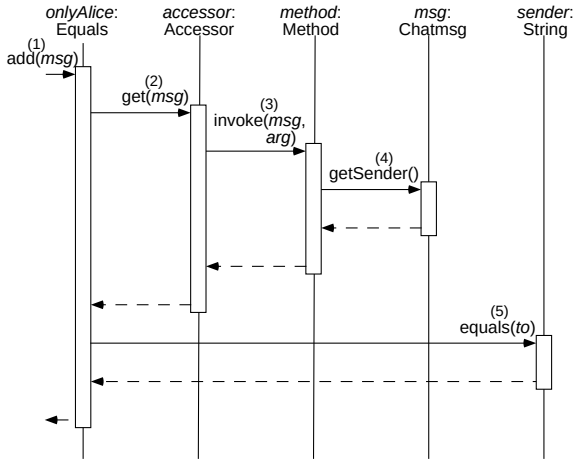

Figure 28. Evaluation of a Condition

Figure 28 shows the runtime matching of the condition object with a message (1). The accessor representing the call of method `getSender()` is evaluated, which results in the invocation of that method (3) on the message object (4). If latter object is not of the class corresponding to the method object, an exception is generated. Otherwise the string returned by the invocation is compared to the value that was given upon the creation of the `Equals` condition object (5).

B.2 Complete Chat Code

In the following we reveal the complete code for the chat example given in the paper. In this extended version, the subscriber is interested only more in what a particular participant says about him/her.

```
1   package DACE.Examples.Chat;
2
```

The `ChatMsg` implements a message that is exchanged between chat participants. The message bears a text stored in a conventional string and a second string representing the sender of the message. It implements the `java.io.Serializable`, which enforces standard java serialization. This class must be in a separate file, in order to allow dynamic invocations of its methods when using reflection in the context of content-based subscribing.

```
3   public class ChatMsg implements java.io.Serializable {
4
5     private String sender;
6     private String text;
7
8     public String getSender() { return sender; }
9     public String getText() { return text; }
10
11    public ChatMsg(String sender,
12                   String text)
13
14      {
15        this.sender = sender;
16        this.text = text;
17      }
18  }
19
```

Class `ChatPublisher` declares a runnable object, which querries the chat participants standard input for messages to publish. It will be run in a dedicated thread.

```
3   import DACE.*;
4   import DACE.Conditions.*;
5   import java.io.*;
6
7   class ChatPublisher implements Runnable {
8
9     private DAC topic;
10    private String sender;
11
12    public ChatPublisher(DAC topic, String sender)
13
14      {
15        this.topic = topic;
16        this.sender = sender;
17      }
18
19    public void run()
20
21      {
22        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
23        for (;;) {
24          try{
25            String toSay = null;
26            toSay = br.readLine();
27            if (toSay != null) {
28              ChatMsg msg = new ChatMsg(sender, toSay);
29              System.out.println(sender + " says " + toSay);
30              topic.add(msg);
31            }
32          } catch(java.io.IOException ioe) {
33          }
34        }
35      }
36  }
37
```

The `ChatSubscriber` class defines the callback object, that will be registered with the DAC, and therefore implements the `Subscriber` interface (Appendix A.2). This subscriber wishes to henceforth only more receive messages from a specific sender (condition `onlyFrom`, line 49) *and* which mention his/her name (`aboutMe`, line 53). Latter condition is expressed by evaluating the method `indexOf()` of the string representing the text. If the string given as argument (the subscribers name) is contained in the text, a value different from -1 is returned.

```
38  class ChatSubscriber implements DACE.Subscriber {
39
40    ChatSubscriber(DAC topic,
41                   String me,
42                   String from)
43      {
44        Condition finalCond = null;
45        Condition onlyFrom = null;
46        Condition aboutMe = null;
47        try {
48          /* We only want messages from 'from' */
49          onlyFrom = new Equals("/getSender", from, true);
50          /* We only want messages concerning ourself, i.e., the text contains 'me'. */
51          /* In other words, the method 'indexOf()' has to return something != -1. */
52          Object[][] params = {null, {me}};
53          aboutMe = new Equals("/getText/indexOf", params, new Integer(-1), false);
54          Condition fromAboutMe = new And(onlyFrom, aboutMe);
55          topic.contains(this, fromAboutMe);
56        } catch (InitializationException iex) {
57          iex.printStackTrace();
58        }
59      }
60
61    public void contains(Object o,
62                         String topicName)
63      {
64        try {
65          ChatMsg msg = (ChatMsg)o;
66          System.out.println("Message from " + msg.getSender() + " : " + msg.getText());
67        } catch (ClassCastException ccex) {
68        }
69      }
70  }
71
```

The `main()` clause of the chat client performs three operations. It first creates a DAC (proxy) for the chat topic given as first argument, before a subscriber object is created, which registers itself with the DAC. Finally, a publisher object is created as well as a thread to execute it.

```
72  public class ChatClient {
73
74    public static void main(String[] args)
75
76      {
77        if (args.length != 3) {
78          System.err.println("ChatClient <topic> <participant> <person of interest>");
79          System.exit(0);
80        }
81        String chatTopicName = new String("/Chat/" + args[0]);
82        DASet chatTopic = null;
83        try {
84          chatTopic = new DAStrongSet(chatTopicName);
85        } catch (InitializationException ie) {
86          ie.printStackTrace();
87          System.exit(-1);
88        }
89        Subscriber chatSubscriber = new ChatSubscriber(chatTopic, args[1], args[2]);
90        publisherThread = new Thread(new ChatPublisher(chatTopic, args[1]));
91        publisherThread.start();
92      }
93  }
```