Content-Based Publish/Subscribe with Structural Reflection*

Patrick Th. Eugster   Rachid Guerraoui
Communication Systems Department
Swiss Federal Institute of Technology, Lausanne
{Patrick.Eugster, Rachid.Guerraoui}@epfl.ch, http://www.d-a-c-e.com

Abstract

This paper presents a pragmatic way of implementing content-based publish/subscribe in a strongly typed object-oriented language. In short, we use structural reflection to implement filter objects through which applications express their subscription patterns. Our approach is pragmatic in the sense that it alleviates the need for any specific subscription language. It preserves encapsulation of message objects and helps avoiding errors. We illustrate our approach in the context of Distributed Asynchronous Collections (DACs), programming abstractions for message-oriented interaction. DACs are implemented in Java, whose inherent reflective capabilities fully satisfy the requirements of our content-based subscription scheme. Our approach is however not limited to the context of DACs, but could be put to work easily in other existing event-based systems.

1 Introduction

Publish/subscribe in perspective. The importance of flexible, well-structured, but especially scalable communication mechanisms has been drastically increasing in the last decade. Applications tend to become very dynamic, i.e., components are not always up and are not locality-bound. These constraints visualize the demand for more flexible communication models, reflecting the nature of tomorrows applications. The *publish/subscribe* interaction style has proven its ability to fill this gap. Based on the concept of *information bus* [OPSS93], publish/subscribe promotes the *decoupling* of parties in *time* as well as *space*:[1] consumers *subscribe* to the information bus by specifiying the nature of the information they are interested in, and producers publish information on that bus.
The classical *topic-based* or *subject-based* publish/subscribe style involves a classification of the information by introducing group-like notions [Pow96], and is incorporated by most industrial strength solutions, e.g., [Cor99, TIB99, Ske98, AEM99]. Topics are however static and allow only a limited *expressiveness* [Car98]. More recently, research efforts have been targeted towards *content-based* (*property-based* [RW97]) publish/subscribe schemes [Car98, SA97, BCM+99]. This more flexible variant removes entirely the "arbitrary" division of the information space, and lets consumers delineate their individual interests by expressing *properties* of messages they wish to receive.

Current practice. Common event-based systems relying on the content-based publish/subscribe paradigm equate *properties* of messages to *attributes* of those messages. In most cases, a subscription language is used to express ranges of values for those attributes, which violates object encapsulation: a subscription *pattern* [2] expressed with such a subscription scheme exposes the message's *state*, and the resulting *filter* queries messages by accessing their *attributes*. Furthermore, subscription languages can not be extended or customized by the application developer, they are orthogonal and redundant with the programming language,[3] and they are very er-

*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] In [Car98], the notion of *pattern* is used in a different sense, namely to express *event-correlation*: a notification is triggered upon arising of a combination of several elementary events.

[3] A similar mismatch has been largely discussed in the domain of object-oriented databases, where two separate languages coexist; one for the *definition* of data and another one for the *querying* of data [BZ87].

ror prone: syntax errors violating the subscription grammar are only seized at runtime when a pattern is parsed, just like syntaxically correct constraints based on badly written attributes.

Filter library. Our approach which has been realized in the context of Distributed Asynchronous Collections (DACs) [EGS00a] – simple Java programming abstractions which encompass different message-oriented interaction styles – avoids any subscription language, and respects encapsulation. It promotes the expression of subscription patterns by combining general-purpose *filter objects*. These filter objects preserve encapsulation by querying message objects through methods which are dynamically defined by the application, along with the semantics of the evaluation of the invocation results. The subscription grammar is inherently expressed through the resulting API, which strongly reduces the number of runtime errors. Filters are thus pictured as first class citizens, and their implementation relies on *structural reflection* [Coi87] of the message objects.

Structural reflection. As pointed out in [Fer89], there are mainly two kinds of reflection. The *computational* (*behavioral*) reflection is concerned with the *reification* of computations and their behaviour. In contrast, *structural* reflection reifies the structural aspects of a program, such as data types.
As we will show in this paper, structural reflection can be used to express subscription patterns in content-based publish/subscribe the same way it has already been used in object-oriented data management systems to express object queries (e.g., [SO95]). In our particular context, structural reflection can be reduced to a single aspect: the capability of *representing* structures of objects. This is sufficient to dynamically define the methods that our filters must use to query message objects. The *introspection* capabilities of Java [Sun99a] offer sufficient support for this, and the possibility of *modifying* data structures is not required.

Contributions. This paper presents how we have realized content-based publish/subscribe in our DAC framework for distributed computing, which is implemented in Java on Unix. We illustrate how our approach (1) circumvents the need for any subscription language, (2) preserves object encapsulation, and (3) helps avoiding type errors. We discuss the flexibility/performance trade-off introduced by our use of reflection by outlining the optimizations we have applied, such as runtime generation of static code from dynamically defined filters.

Roadmap. This paper is structured as follows: Section 2 overviews the limitations of existing approaches to expressing content-based subscription patterns. Section 3 presents our approach to content-based publish/subscribe based on structural reflection. In Section 4 we illustrate the use of our subscription scheme through a small example. Section 5 discusses performance issues. Section 6 highlights alternative approaches. Section 7 concludes with final remarks.

2 Approaches to Content-Based Publish/Subscribe: Background

Content-based publish/subscribe removes limitations of the static topic-based flavor, but suffers from the dynamism it introduces. Besides making the reuse of existing multicast primitives problematic [OAA+00], content-based publish/subscribe is hard to express in an object-oriented setting. In this section, we illustrate the latter difficulty by outlining the limitations of existing content-based schemes.

2.1 Subscription Languages

In content-based publish/subscribe, subscription languages are the most commonly used means of describing subscription patterns. Such languages can be based directly on the *attributes* of the described objects or on additional *properties* attached to those objects. By viewing asynchronous invocations as events, the *arguments* of such invocations can be used as matching criteria.

Attributes. In systems like Siena [Car98], Elvin [SA97] or Gryphon [BCM+99, SBS98],[4] event notifications are viewed as flat structures,

[4] In Gryphon, reflection is also used ([SBS98]), but not for the expression of subscription patterns: the Gryphon system uses the same information dissemination mechanisms

i.e., *records* with several *fields*. A subscription language is used to impose ranges of values for those fields. Figure 1 outlines this concept schematically. Relying on *attribute-value* pairs enables very efficient realizations, since computational overhead is reduced by directly accessing attributes. This approach however bears several dangers:

Violation of object encapsulation: In the example outlined in Figure 1, the `from` attribute is used as subscription criterion, and is consequently directly accessed when the object is queried.

Errors: Syntax errors violating the subscription grammar are only seized once a pattern is parsed, i.e., at runtime. Another more malign type of errors result from badly typed attribute names. Subscription patterns containing such errors do not violate the syntax grammar, and might remain undetected without type checks.

Learning phase: Subscription syntaxes are often very complex and used with a single publish/subscribe middleware. This reduces portability of applications.

To increase portability of applications some engines implement standardized API's like the OMG's CORBA NOTIFICATION SERVICE [OMG00], which repairs certain lacks [SV97] of the CORBA EVENT SERVICE [OMG98]. Among the new features in [OMG00] are a content-based subscription scheme based on a simplified kind of typed events, replacing the typed events of the ancestor. These *structured events* are roughly composed of two types of fields, namely (1) fixed fields and (2) variable fields consisting of *name-value* pairs, to which applications map their specific needs. The fields of messages are seen as their attributes and are directly accessed through *filter objects* for content-based filtering – violating encapsulation. Patterns are expressed by strings following the DEFAULT FILTER CONSTRAINT LANGUAGE, a complex subscription language which extends the TRADER CONSTRAINT LANGUAGE.

Properties. The SUN counterpart to the CORBA NOTIFICATION SERVICE is the JAVA MESSAGE SERVICE (JMS) API [HBS98]. The JMS covers topic-based publish/subscribe (*all-of-n*) as well as *message queuing* (*one-of-n*) [BHL95, DEC94, Sys00, Mic97]. Content-based filters can be applied with both interaction schemes. The filtering is based on attributes of the message headers, and on *properties* (name-value pairs), which are explicitly attached to message objects. Subscription patterns are expressed as JAVA strings. The specification includes a subscription grammar that these strings must respect.
Properties explicitly attached to message objects are artificial and in practice strongly redundant with the information carried by those objects. In many cases, the properties are faithful duplicates of the attributes of the message objects, which leads to violating encapsulation.

it offers to applications (which is its primary concern) for internal protocol communication.

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

Criteria	"message sent by Tom"
Argument	`String criteria = "from is Tom";`
Evaluation	`m.from.equals("Tom")`

Figure 1: Subscription Language

Arguments. MICROSOFT's COM+ [Obe00] promotes a model similar to the abandoned typed model of the CORBA EVENT SERVICE. Asynchronous invocations are viewed as events, but latter ones are not reified. The primary filtering is thus made on the types of the subscribers, as illustrated by Figure 2. By viewing an invocation as an event, the invocation arguments can be viewed as the attributes of the resulting notification. Filters in COM+ are expressed on invocation arguments through a limited subscription grammar. Encapsulation seems to be preserved by avoiding the reification of events.

Subscriber s	`public class Chatter {`
	` public void in(String from, ...);`
	` ...`
	`}`

Criteria	"message sent by Tom"
Argument	`String criteria = "from is Tom";`
Evaluation	`from.equals("Tom")`

Figure 2: Events vs. Invocations

2.2 Template Objects

The JavaSpace specification [Sun99b] (inspired by Linda's Tuple Space [Gel85]) adopts an approach based on *template objects*.
When subscribing to a JavaSpace, a subscriber provides a template object `t`. A message object `m` is only delivered to that subscriber if `m` conforms to the type of `t`, and if every attribute of `t` which is not `null` references an object equal to the corresponding attribute of `m` (cf. Figure 3). Equality is tested by comparing byte-wise the two objects in marshalled form. As shown by [FHA99], this approach represents a very convenient subscription scheme which can be put to work easily. However, encapsulation is violated, and there are certain limitations in expressiveness:

Limited comparisons: Attributes are compared for strict equality, and it is not straightforward to express a range (discrete or not) of possible values for an attribute.

Limited granularity: In Java, an attribute can reference an object, which itself has attributes, etc. Attributes of JavaSpace entries are however matched as a whole. This limitation is also found with most of the previous approaches based on subscription languages.

Limited combinations: By providing a template object `t`, a subscriber will receive every object `m` whose attributes *all* match the attributes of `t`. It is thus difficult to express alternatives (*or*) on different attributes.

Limited values: Since `null` is chosen to play the role of wildcard, attributes can not be of native types, and `null` can not be easily used as a concrete value for an attribute. For each such attribute [Sun99c] proposes to add an additional boolean attribute to indicate a `null` value.

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

Criteria	"message sent by Tom"
Argument	`ChatMsg mt = new ChatMsg();` `t.from = "Tom";`
Evaluation	`m.from.equals(t.from)`

Figure 3: Template Object

3 Reflection-Based Publish/Subscribe

In this section we present our approach to specifying content-based subscription patterns. It is based on structural reflection of message objects and avoids limitations stated in the previous section.

3.1 Overview

Roughly spoken, the application programmer *defines* conditions on message objects, by specifying *methods* through which these objects should be queried, along with expected *values* that are *compared* to the values returned by invoking these methods.
Subscription patterns are expressed through an API, which inherently expresses a subscription grammar: by instantiating and combining filter objects, syntax errors violating the grammar are detected by the Java compiler. Thanks to the structural reflection of message objects, type errors are avoided by verifying the methods specified by the application.
In the following, we present our customizable filter objects, called *conditions*. These enable the dynamic definition of conditions on message objects, and are realized in a general manner through *accessors*, which we introduce first.

3.2 Accessors

Accessors are specific objects used to access partial information on the runtime message objects.

Querying objects. Informally, an accessor A is characterized by a set of tuples: $A = ((M_1, P_1), ..., (M_k, P_k))$, where every M_i is a method and $P_i = (P_{i,1}, ..., P_{i,i_l})$ its corresponding argument list. Whenever a method M_i is applied to an object, this subsumes that it is invoked with its arguments P_i.
An accessor can be seen as a function, which applied to a message object returns another object: $A(o : obj) \rightarrow obj$. When such an accessor A is evaluated for a message object m, M_1 is invoked on m

and every method M_{i+1} $(0 < i < k)$ is recursively invoked on the result of M_i. Finally, the result of M_k is returned.[5]

In JAVA, an accessor object implements the interface `Accessor` given in Figure 4, and is evaluated by calling the `get()` method with the message object as argument. This method can throw exceptions raised when evaluating the method chain, which enables the reaction to exceptions. Returning `null` in case of exceptions would contradict the use of `null` as matching criterion.

```
public interface Accessor {

  public Object get(Object m) throws Exception;
}
```

Figure 4: `Accessor` Interface

Using Java reflection. To implement our accessors, we rely on structural reflection. The inherent JAVA language reflection capabilities [Sun99a] consist in a type-safe API that supports *introspection* about classes and objects in the current JAVA VM at runtime. We view introspection as one aspect of structural reflection, limited to the reification (in the sense of *representation*) of *structures* of types and classes at runtime. A second aspect, the *modification* of those structures is, like computational reflection, not addressed by the JAVA core reflection API.[6]

In short, JAVA provides *meta-objects* which reify classes, methods, fields, constructors, etc. We make extensive use of meta-objects for methods (`java.lang.reflect.Method`) to reify the M_i's of accessors. This defers to runtime the choice of *which* method is to be invoked, and enables also to effectively perform such a *dynamic invocation*.[7] We avoid using objects reifying attributes (`java.lang.reflect.Field`), since dealing with them means abandoning encapsulation.

[5] With $k = 0$, the object m itself is accessed as a whole. $i_l = 0$ means that M_i is an argument-less method. We do not consider side-effects of the access methods M_i.

[6] JAVA 1.3 integrates a limited mechanism for computational reflection with the `java.lang.reflect.Proxy` class.

[7] Note that with JAVA method objects, a native value is wrapped by an instance of its corresponding object type, which makes the nesting of invocations even simpler.

```
public final class Invoke
  implements Accessor, java.io.Serializable
{
  /* only one method, can be null */
  public Invoke(Method M, Object[] args) {...}
  /* with nested accessor */
  public Invoke(Accessor nested, Method M,
                Object[] args) {...}
  /* structurally conformant objects, nesting */
  public Invoke(String methodNames,
                Object[][] args) {...}
  ...
  public Object get(Object m)
                  throws Exception {...}
}
```

Figure 5: `Invoke` Class (Excerpt)

Specifying methods. We have used JAVA reflection for the implementation of the `Invoke` class shown in Figure 5, a general-purpose accessor. The first constructor enables the expression of a single method invocation. The other constructors shown in the figure enable the creation of an accessor reflecting nested method calls; by specifying an explicitly created nested accessor, or by specifying the names of the methods to be invoked. This adduces the two ways for an application to specify a method:

By method object: The application explicitly deals with reflection, and provides a `Method` object. As explained in [BW98], JAVA enforces *name equivalence* of types, and a method object `M` is therefore bound to a single type `T`: if a method `M` for type `T` is applied to an object `m` which does not conform to `T`, `null` is returned – even if `m` implements a method of the same name and signature than `M`. By specifying methods as objects, the application implicitly defines the type of message objects it is interested in.

By method name (and signature): Specifying the name of a method and its arguments (and implicitly the method's signature) can be interesting to enforce *structure equivalence* of types, i.e., subscribing to *all* objects which implement a given method, independently of their type. This implies, for *each* evaluated message object, a *dynamic lookup* of the corresponding method object (through `java.lang.Class`) by the accessor.

In Section 5 we evaluate the two possibilities in terms of efficiency, and show that the knowledge of the type of message objects is important for performance optimizations.

Avoiding type errors. Knowing the type of the fitting message objects is also useful for type checking. If all methods of an accessor are reified, the return type of each such M_i can be checked for its conformance to the type bound to M_{i+1}. Similarly, the type of each provided argument $P_{i,j}$ can be checked for its conformance to the type of the j-th formal argument of M_i. By enforcing these checks, the `Invoke` class rules out type errors.[8] To enforce such checks without explicit use of reflection, message object types can also be specified by their name. This is illustrated in Section 4 through a small programming example.

3.3 Conditions

While a message object is queried through an accessor, a condition object evaluates the obtained information, i.e., decides whether it represents a desirable value.

Model. A condition $C = (A, R, B)$ represents a single condition that a message object m must fulfill in order to be delivered. B is a comparison function which can be viewed as a binary predicate: $B(o_1 : obj, o_2 : obj) \to bool$. The two arguments are (1) a predefined result R and (2) the result of the invocation chain represented by the accessor A. A condition is thus evaluated against a message object m, and evaluates positively iff m satisfies that condition: $C(o : obj) \to bool$, and $C(m) = B(R, A(m))$. Figure 6 outlines the different evaluation stages of a condition. A similar scheme can be found for object queries in object-oriented data management systems, e.g., TIGUKAT [SO95].[9]

[8]To verify whether a given reified type conforms to another one, we mainly rely on the `isAssignableFrom()` method in class `java.lang.Class`.

[9]The major difference between queries in an object database and the filtering of messages by a middleware is the *duration* of a query. With a middleware system based on content-based publish/subscribe, the query is expressed for *future* objects. In object databases, queries are performed on a snapshot of the system, but the expression of the query can be made similarly. [PO93] also describes the use of reflection for a closer integration of the language with TIGUKAT.

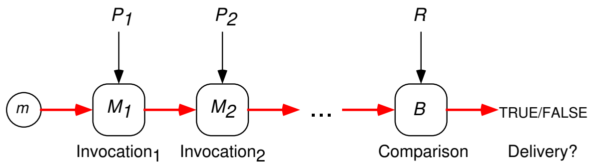

Figure 6: Applying a Condition to a Message Object

Basic conditions. In JAVA, a condition object implements the `Condition` interface given in Figure 7. It is evaluated for a given message object `m` by invoking `conforms()` with `m` as argument. The condition classes we propose are conceptually similar to the *predicates* found in JGL [Obj99] that are used in conjunction with centralized collections. The main difference is that our condition objects are specific to publish/subscribe, by representing queries on *future* objects.
What differentiates our condition classes are the comparison functions they encapsulate. The other attributes, namely accessor and result, are initialization arguments and can thus be factored out.

```
public interface Condition {

  public boolean conforms(Object m);
}
```

Figure 7: `Condition` Interface

Comparisons. JAVA inherently defines three basic comparison mechanisms, which can be considered as candidates for B:

I. **Is object `o1` identical to object `o2`?**
The comparison of two objects with the `==` operator yields `true` iff the two arguments are references to the same object. This comparison is less useful in our context, since two compared objects usually originate from different VMs. By default two such objects are never identical.

II. **Is object `o1` equal to object `o2`?**
Every object can also be compared to any other object by means of the `equals()` method, which is inherent to all JAVA objects and can be overwritten by application-defined classes.

III. **How does object `o1` compare to object `o2`?**
This is for objects implementing the `java.lang.Comparable` interface,[10] providing a method `compareTo()`. The return value is an integer, indicating the order of the object `o1` with respect to `o2`. Such objects manifest a natural ordering, e.g., class `java.lang.Integer`, and can thus be matched against a range of values. Comparisons can be moved out of the compared objects by using `java.util.Comparator` objects, which are binary predicates.

In general, B is represented in JAVA by a method, and can also be viewed as M_{k+1}. Inversely, methods $M_j ... M_k$ ($j \geq 1$) can be seen as part of the comparison. In that sense, we provide several shortcuts for common methods, e.g., to compare the type of an object to a predefined one. This reflects the method `isInstance()` (the dynamic counterpart to the `instanceof` operator) in `java.lang.Class`.
In our condition classes, like the `Equals` class given in Figure 8 (representing an equality test in the sense of II), we have added constructors which alleviate their use. The third constructor in the figure for instance enables the expression of nested method calls by providing a URL-like string denoting the names of the methods. The accessor is in that case created implicitly. Figure 9 shows the links between our JAVA implementation of accessors and conditions, illustrated through the `Equals` and `Invoke` classes.

```
public final class Equals
  implements Condition, java.io.Serializable
{
  /* compare the message object as a whole */
  public Equals(Object to) {...}
  /* compare return value of accessor */
  public Equals(Accessor acc, Object to) {...}
  /* implicit accessor creation */
  public Equals(String names, Object[][] args,
                Object to) {...}
  ...
  public boolean conforms(Object m) {...}
}
```

Figure 8: `Equals` Class (Excerpt)

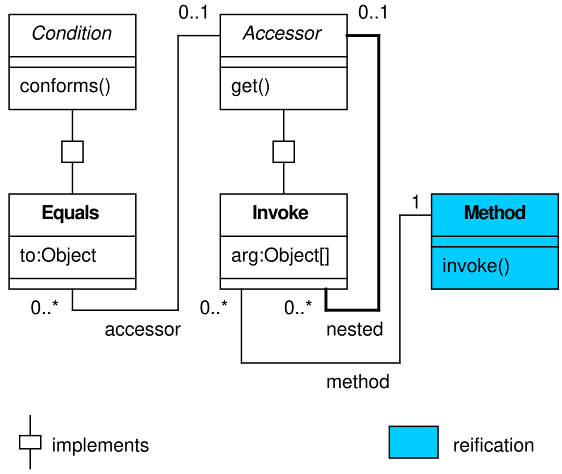

Figure 9: Class Diagram

3.4 Subscription Patterns

A subscription pattern S represents a combination of basic conditions.

Patterns and conditions. A subscription pattern $S = ((C_1, ..., C_n), F)$ is characterized by a set of n basic conditions, which are all evaluated for a given message m, and a n-ary function $F(b_1 : bool, ..., b_n : bool) \rightarrow bool$, which is evaluated for the results of these conditions: $S(m) = F(C_1(m), ..., C_n(m))$. A pattern is thus evaluated like a condition: $S(o : obj) \rightarrow bool$, and is represented in JAVA by an object of type `Condition`. The model diverges here from the concrete realization, in that the function F does not appear as such.

Expressing patterns. F is namely explicitly constructed by *combining* conditions. These combinations are expressed through specific conditions, reflecting binary predicates, like `And` (Figure 10), `Or`, etc. Furthermore, we propose a condition `Not` for negation. To ease the expression of combinations, we introduce the `SimpleCondition` interface (Figure 11), an extension of `Condition`, which our basic conditions in fact implement.[11]

This subscription scheme based on conditions inherently expresses the subscription grammar. Syntax

[10] The counterpart to the well-known `Magnitude` type in SMALLTALK [GR83].

[11] This counteracts JAVA's lack for operator overloading (as provided for instance by C++ [Str97]).

errors known from subscription languages, where they are only recognized at execution of the parser, are here detected by the JAVA compiler.

```
public final class And
  implements Condition, java.io.Serializable
{
  /* the two arguments */
  private Condition first;
  private Condition second;

  public And(Condition first,
             Condition second)
    { this.first = first; this.second = second; }

  public boolean conforms(Object m)
    { return first.conforms(m) &&
        second.conforms(m); }
  ...
}
```

Figure 10: `And` Class (Excerpt)

```
public interface SimpleCondition
  extends Condition

{
  public SimpleCondition and(Condition with);
  public SimpleCondition or(Condition with);
  public SimpleCondition nand(Condition with);
  public SimpleCondition nor(Condition with);
  public SimpleCondition xor(Condition with);
  public SimpleCondition not();
}
```

Figure 11: `SimpleCondition` Interface

4 Programming Example

This section illustrates the use of content-based filters through *chat sessions* based on simple DACs. We first recall our notion of DISTRIBUTED ASYNCHRONOUS COLLECTION (DAC) and then build on the example initially introduced in [EGS00a].

4.1 Background: Distributed Asynchronous Collections

Just like a conventional collection, a DAC represents a group of related objects. A DAC is however distributed and can be accessed from various nodes of a network. In a way similar to a JAVASPACE, a DAC enables distributed participants to share information by pulling information from the space, but also by registering a callback object to be notified of future elements. DACs furthermore express through their type the *qualities of service* (QoS) they support. In other terms, we offer a framework of DAC subtypes representing different semantics and QoS. In this example, we will use a DAC representing a topic, to which application components subscribe with an optional content-based filter.

4.2 A Chat Scenario

We concentrate on two chat addicts, Alice and Bob, who love to chat deep into the night. Therefore they subscribe to the topic */Chat/Insomnia* to receive all messages from like-minded chatters. Figure 12 shows class `ChatMsg`, which represents a possible message class for this application.

```
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 12: Event Class for Chat Example

4.3 Publishing

A DAC represents a topic, and in order to access such a DAC from a process, a proxy must be created. This requires an argument denoting the name of the topic represented by the DAC. Except for that topic name, the action of creating a DAC proxy is identical to creating a local collection. The `DAStrongSet` collection class instantiated in Figure 13 offers reliable delivery of notifications to subscribers. The instance called `myChat` henceforth provides access to 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, from where it becomes accessible for any party.

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create new message and publish it */
ChatMsg m = new ChatMsg("Alice", "Hi everyone");
myChat.add(m);
...
```

Figure 13: Publishing a Message

4.4 Subscribing

In order to subscribe to a DAC, a callback object implementing the `Notifiable` interface must be provided. Figure 14 shows how to implement a simple callback object for chat sessions.

```
public interface Notifiable {

  public void notify(Object m);
}

public class ChatNotifiable
  implements Notifiable
{
  public void notify(Object m) {
    /* elements are of type ChatMsg */
    ChatMsg cm = (ChatMsg)m;
    System.out.println("Message from " +
                       cm.getSender());
    System.out.println(cm.getText());
  }
}
```

Figure 14: Callback Object

Episode I. Figure 15 shows a first example of content-based publish/subscribe. We suppose here that Bob is only interested in what a particular participant, Alice, publishes on topic */Chat/Insomnia*. Bob defines a corresponding condition. The `null` argument in the condition initialization denotes a set of empty argument lists. A subscription is viewed as an interest in future elements, and is expressed by a call to the `contains()` method.

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create condition */
Condition onlyAlice =
  new Equals("/getSender", null, "Alice");

/* create callback object and subscribe */
Notifiable n = new ChatNotifiable();
myChat.contains(n, onlyAlice);
...
```

Figure 15: Content-Based Subscribing (I)

Episode II. Suppose now that Bob is only more interested in what Alice says about him. For this second condition, the text carried by each chat message must be checked for the occurrence of Bob's name. Remember that in JAVA a string `s1` can be checked for the occurrence of a substring `s2` by asking `s1` through a call to `indexOf()` for the index of its first occurrence of `s2`. If `s2` is not contained in `s1`, the call returns `-1`. The resulting second condition in the figure represents all messages which do *not* contain Bob's name, and must therefore be negated.
This example shown in Figure 16 illustrates how to easily combine basic conditions with the `SimpleCondition` interface, and how the application can specify the type of the message objects with implicit accessor creation, as required for the performance optimizations we propose in the next section.

5 Performance

Reflective systems and meta-level architectures offer increased modularity and flexibility. The benefit of such dynamism is often, but not necessarily, diminished by performance degradation. In this section we first give a rough idea of the cost of dynamic code introduced by JAVA reflection. Motivated by these results we then propose two optimizations to our system, and we discuss their performances.

5.1 Preliminary: Cost of Reflection

According to the way we have described our implementation, methods are invoked dynamically, i.e.,

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create first condition, with type specification */
SimpleCondition onlyAlice =
  new Equals("ChatMsg:/getSender", null, "Alice");

/* create args list and corresp. second condition */
Object[][] args = {{null}, {"Bob"}};
SimpleCondition notAboutMe =
  new Equals("ChatMsg:/getText/indexOf", args,
             new Integer(-1));

/* combine conditions */
SimpleCondition pattern =
  onlyAlice.and(notAboutMe.not());

/* create callback object and subscribe */
Notifiable n = new ChatNotifiable();
myChat.contains(n, pattern);
...
```

Figure 16: Content-Based Subscribing (II)

through reified methods. Such dynamic invocations are much more expensive than static ones. Moreover, when subscribing to structurally conformant objects (cf. Section 3), method objects are obtained at runtime for each message object. Such lookups are very costly, and are summed with the overhead of the dynamic invocations.

Figure 17 shows the cost of dynamic calls by comparing the overhead of local method invocations with a varying number of arguments (between 0 and 10 objects). These are performed using (1) dynamic invocations, each combined with a method lookup, (2) dynamic invocations without lookups, and (3) static invocations. These tests were made on a Sun Ultra 60 (Solaris 2.6, 256 Mb RAM, 9 Gb harddisk) with Java 1.2 (native threads). The test setting did not involve any Just In Time (JIT) compiler. The speedup factor observed for static invocations when using a JIT compiler was over three. The speedup in the case of dynamic evaluation is, as expected, insignificant.

5.2 Optimizations

The amount of expensive dynamic code can be reduced if the type of the message objects is known. The type information can be given to the system either (1) by using reflection explicitly, or (2) by specifying the type of the message objects by name.

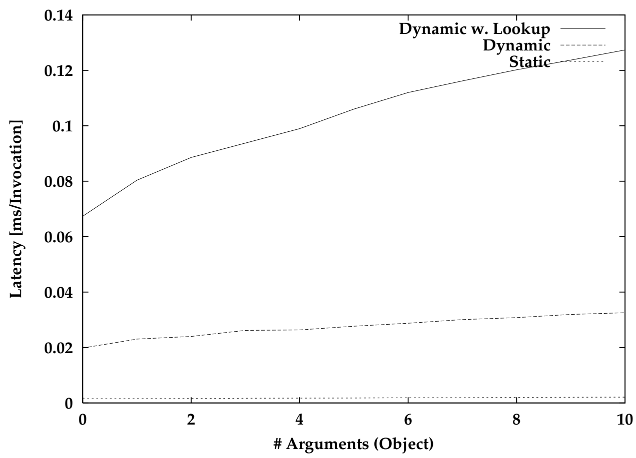

Figure 17: Latency with Different Invocation Styles

The type information enables the application of optimizations.[12]

Avoiding redundant invocations. Message objects are usually matched against patterns of several subscribers at a time, and these patterns are likely to present redundancies. We discuss here an optimization based on that observation, which is similar, but not identical to the *tree matching* algorithm used in Gryphon [ASS+98]. The tree matching algorithm factors out redundant subpatterns with simplified assumptions: only *and*s of basic conditions are considered, and latter ones are primitive comparisons of attribute values with predefined values.

In contrast, our filter library offers more expressiveness, e.g., nested method invocations, different comparators and combinations (*and*, *or*, ...). Such combinations are performed statically, and dynamic queries on message objects represent the critical factor in our system. As a consequence, we focus on detecting common denominators of accessors, in order to avoid the evaluation of redundant dynamic method invocation chains. Figure 18 shows a simple example of redundant accessors where each subscriber specifies a pattern consisting of a single basic condition. An *invocation tree*, like the one shown in Figure 19, is constructed from all accessors and is evaluated for every filtered message object.

[12] In a companion paper [EGS00b] we introduce *type-based* publish/subscribe: a static classification scheme based on the *types* of message objects. The *type-based* publish/subscribe scheme ensures type safety, and thus enforces optimizations through the inherent knowledge of types and makes type

Subscriber S_1: $A_{1_1} = ((M_1, P_1))$
Subscriber S_2: $A_{2_1} = ((M_2, P_2), (M_3, P_3))$
Subscriber S_3: $A_{3_1} = ((M_2, P_2), (M_3, P_3), (M_4, P_4))$
Subscriber S_4: $A_{4_1} = ((M_2, P_2), (M_3, P_5))$

Figure 18: Redundancy between Accessors

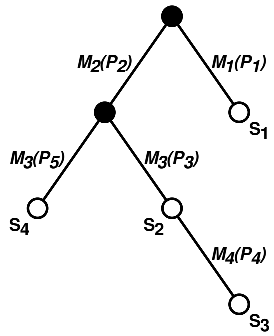

Figure 19: Invocation Tree

Enforcing static filters. Based on the observation that dynamic invocations are far more costly than static ones, we have implemented an alternative optimization. Any dynamic invocations are avoided by generating static source code from accessors after performing type checks. The source code is then directly compiled by calling the SUN JAVA compiler (`sun.tools.javac`), in a way similar than this is done in [KMS98] or [TCKI00].[13]

5.3 Evaluation

We evaluate here the benefits of the two above optimizations by comparing the resulting performances with two non-optimized scenarios. These are namely (1) the filtering of structurally conformant messages, and (2) the filtering of type conformant messages.

Testbed. Our measurements were made with the JAVA VM 1.2, enabled JIT and native threads on SUN SOLARIS 2.6. A single producer was publishing message objects encapsulating a single string from one network (SUN ULTRA 60, 256 Mb RAM, 9 Gb harddisk), to subscribers equally distributed over two further networks; one composed of all together 60 SUN SUPERSPARC 20 stations (model 502: 2 CPU, 64 Mb RAM, 1Gb harddisk), and the second one composed of 60 SUN ULTRA 10 (256 Mb RAM, 9 Gb harddisk) stations. The individual stations and the different networks where communicating via FAST ETHERNET.

Parameters. We have made a set of extensive tests, in which we have always varied one of four parameters for the subscriptions. These are namely, (1) the fraction of positive matches for a subscriber $1/c$, (2) the total number of subscribers s, (3) the maximum nesting level of invocations for queries a, and (4) the number of different query methods d at each nesting level.

Varying $1/c$*:* From n produced messages, an average of n/c messages matched a given subscribers pattern. Figure 20(a) shows the effect of varying c. It confirms the intuition that the cost of sending messages with UDP does not depend on the matching scheme, and can be seen as fixed. With $c > 100$ in this scenario, the pure cost of matching is measured. In order to accentuate the differences between the matching schemes without contradicting our concrete applications, we have chosen $c = 10$ for the next figures.

Varying s*:* Similarly to the scenario in Figure 18, we have chosen one basic condition per subscriber. Figure 20(b) reports the effect of scaling up s, conveying that the two optimizations are almost equivalent with a large s.[14] As shown in the previous figure, UDP is a limiting factor with an increasing number of sends (here due to a large s). Performance drops faster with static filters, since every additional subscriber involves a full pattern evaluation. In contrast, the optimized dynamic scheme is less sensitive since redundant queries are avoided.

Varying a*:* The probability of having $i \in [0, a]$ nested invocations was chosen as $p_a = 1/(a + 1)$. Increasing a reduces throughput with static invocations (Figure 21(a)), since static accessors comprise more invocations. Similarly, the optimized

checks and casts inside the application code superfluous.

[13][KMS98] terms this technique (runtime) *linguistic* reflection, which is seen as a synonym of *structural* reflection.

[14]Our system relies on a hierarchical topology of message brokers, among which membership information is split up. A single process rarely has knowledge of more than 100 participants.

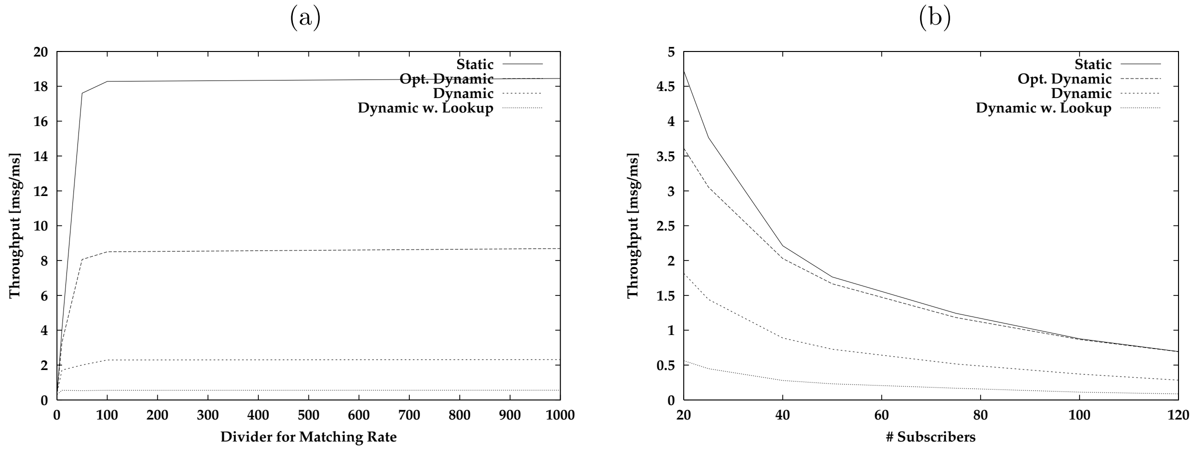

Figure 20: Matching Rate and Number of Subscriptions

dynamic scheme is less efficient with an increasing a, since the total number of performed methods increases with the depth of the tree.

Varying d: One of d methods was chosen at each nesting level with a probability of $p_d = 1/d$. Varying d obviously does not influence static filter evaluation. On the other hand, increasing d might lead to increasing the potential number of edges leaving from any node in the invocation tree. The resulting performance loss is directly visible in Figure 21(b). The optimized dynamic scheme is however more penalized by increasing a, as shown in the previous figure. This is due to the fact that increasing a by 1 might result in up to d new edges in every former leaf of the invocation tree.

Interestingly the optimized dynamic matching scheme never overperforms the static scheme, even if the speedups become close with a large number of message sends. One could believe that with a strong redundancy between patterns and a large number of subscribers the dynamic scheme would become more efficient. Even with extreme parameter values, we have however never encountered such a scenario.

6 Discussion

In this section we debate alternative models and realizations we have considered as potential solutions for an adequate content-based subscription scheme in the context of Distributed Asynchronous Collections.

Application-defined filters. We promote the expression of subscription patterns as a combination of instances of our predefined condition classes. An alternative to this consists in allowing the application to provide directly its own static filter objects (byte code). Patterns expressed this way are however opaque and not necessarily correct nor safe, and make optimizations difficult.
Nevertheless, we have opted for an open design, i.e., separation of interfaces and classes (e.g., `Accessor/Invoke`) vs conditions and accessors as `final` classes. This enables the extension of our subscription API with application-defined accessors and conditions. Our proposed optimizations can still be enforced by following certain design guidelines.

Towards a unified language. An alternative to our subscription API consists in using the Java language itself as the subscription language. That is, providing code in a stringified form (source code), that can be parsed and compiled at runtime. A pseudo-variable `m` would represent the runtime message object, and method invocations could be directly expressed, e.g.:

```
"m.getSender().equals("Alice") && ...;"
```

The evaluation of the code given here as a string is *deferred*. This comes to introducing two levels of programming, in a way similar to [NN88]. The generalization of that approach leads to *multi-stage*

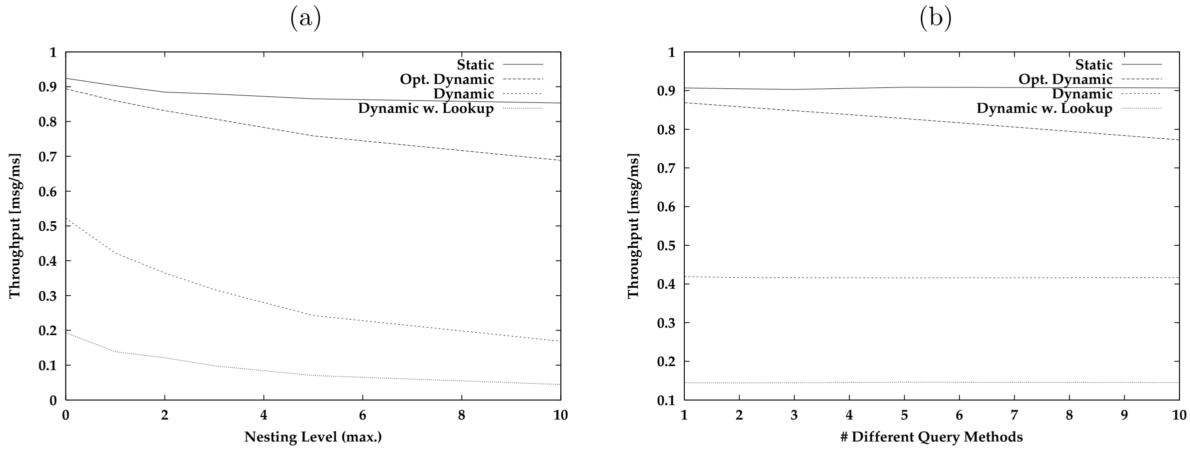

Figure 21: Expressiveness and Redundancy of Subscriptions

programming, e.g., MetaML [TS97]. MetaML is a meta-programming language that was designed as a homogenous runtime code generator toolkit. In MetaML, the evaluation of expressions in <> (called brackets) is deferred to the next stage, in a sense similar to our stringified code delimited by `""` above. Expressions evaluated at a later stage can refer to constructs at a previous stage. When stringifying meta-code in Java as above, this is not possible, since Java reflection does not allow to dynamically obtain a reference to a variable by its name. This limitation can be circumvented as long as invocation arguments can be constructed inside the pattern string (e.g., `"Alice"`), but poses problems for complex matches. Extending the Java language in the sense of MetaML would have contradicted our resolution of using merely standard language constructs.

Java reflection. Javassist [Chi00] and OpenJava [TCKI00] are two approaches to extending Java with load-time structural reflection, i.e., the ability of *modifying* classes at runtime prior to instantiation. OpenJava promotes a compile-time meta-object protocol [Chi95] based on an extension of `java.lang.Class`, and makes use of the Sun Java compiler, while Javassist provides an extended `Classloader` supporting the creation of new methods as copies of existing ones.

We have however refrained from using Javassist or OpenJava, because our static filters represent very specific classes which can be generated without any language extension.

7 Concluding Remarks

We argue through our work that, unlike what is often claimed (e.g., [Koe99]), message-oriented middleware and object-oriented principles are not contradictory. In [EGS00a], we have made a first step, by introducing a programming abstraction called Distributed Asynchronous Collection (DAC) which is versatile enough to express commonalities between the different message-oriented interactions styles. In that paper we have focused on topic-based publish/subscribe.

In this paper, we have attacked another bastion, content-based publish/subscribe, which is presumed to contradict object-oriented principles by its very nature. We have illustrated that it is indeed possible to express content-based subscription patterns in a way that fully preserves encapsulation. Moreover, we have shown that our approach offers further practical benefits over contemporary approaches, like the possibility to prevent syntax errors and type errors.

In terms of performance, the cost of our solution is incurred by the latency resulting from the *use* of Java reflection. In our case, this use is however reduced to verifications of subscription patterns aiming at avoiding type errors. After this initialization phase, dynamic invocations are circumvented by using static code generated at runtime without any modification to the Java compiler or VM. The throughput of our system is thus not conditioned by the use of reflection, as proven by the resulting performances. We are furthermore currently working on a new optimization scheme combining the benefits of our static and dynamic optimizations. The

idea is to generate static code from dynamic invocation trees, to further improve performance but also to reduce the overall compilation effort.
The cost of our solution in terms of feasibility is limited to the *need* for structural reflection; yet with such minimal features that the inherent JAVA reflection capabilities can satisfy this need.
We do not claim that our content-based subscription scheme is the ultimate solution to content-based publish/subscribe, nor that it replaces existing specifications. It should rather be seen as a pragmatic attempt to circumventing shortcomings of other approaches. Our filter library is not limited to the context of DACs, but could be put to work easily in other existing event-based systems.

Acknowledgments

We would like to thank both Andrew Black and Joe Sventek for their valuable comments. Those comments have helped us concretize the ideas presented in this paper.

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 of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS'99)*, pages 43–53, October 1999.

[ASS+98] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, November 1998.

[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.

[BHL95] B. Blakeley, H. Harris, and J.R.T. Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development.* McGraw-Hill, 1995.

[BW98] M. Büchi and W. Weck. Compound types for Java. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 362–373, October 1998.

[BZ87] T. Bloom and S.B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 441–451, 1987.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, December 1998.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 285–299, October 1995.

[Chi00] S. Chiba. Loadtime structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, pages 313–336, June 2000.

[Coi87] P. Cointe. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 156–167, October 1987.

[Cor99] Talarian Corporation. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper).* http://www.talarian.com/, 1999.

[DEC94] DEC. *DECMessageQ: Introduction to Message Queuing*, April 1994.

[EGS00a] P.T. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, pages 252–276, June 2000.

[EGS00b] P.T. Eugster, R. Guerraoui, and J. Sventek. Type-based publish/subscribe. Technical Report DSC/2000/029, Swiss Federal Institute of Technology, Lausanne, http://dscwww.epfl.ch/EN/publications/, June 2000.

[Fer89] J. Ferber. Computational reflection in class based object-oriented languages. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, pages 317–326, October 1989.

[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 (TOPLAS)*, 7(1):80–112, January 1985.

[GR83] A.J. Goldberg and A.D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

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

[KMS98] G. Kirby, R. Morrison, and D. Stemple. Linguistic reflection in java. *Software - Practice and Experience*, 28(10):1045–1077, 1998.

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

[Mic97] Microsoft. *Microsoft Message Queuing Services*, 1997.

[NN88] F. Nielson and H.R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[OAA+00] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in content-based publish-subscribe systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, pages 185–207, April 2000.

[Obe00] R.J. Oberg. *Understanding & Programming COM+.* Prentice Hall, 2000.

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

[OMG98] OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service.* OMG, December 1998.

[OMG00] OMG. *Notification Service Standalone Document.* OMG, June 2000.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principles*, pages 58–68, December 1993.

[PO93] R.J. Peters and M.T. Özsu. Reflection in a uniform behavioral object model. In *Proceedings of the 12th International Conference on Entity-Relationship Approach*, pages 37–49, 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 *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, pages 344–360, 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 (AUUG'97)*, http://www.dtsc.edu.au/, September 1997.

[SBS98] D.C. Sturman, G. Banavar, and R. Strom. Reflection in the Gryphon message brokering system. In *Reflection Workshop of the 13th 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.

[SO95] D.D. Straube and M.T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), April 1995.

[Str97] B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Sun99a] Sun. *Java Core Reflection API and Specification*, 1999.

[Sun99b] Sun. JavaSpaces specification. Technical report, Sun Microsystems Inc., November 1999.

[Sun99c] Sun. Jini Entry specification. Technical report, Sun Microsystems Inc., November 1999.

[SV97] D. Schmidt and S. Vinoski. Overcoming drawbacks in the OMG Event Service. *SIGS C++ Report magazine*, 19(6), June 1997.

[Sys00] BEA Systems. *Reliable Queuing Using BEA Tuxedo: White Paper.* http://www.beasys.com/products/, 2000.

[TCKI00] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. *A Class-based Macro System for Java*, pages 119–135, LNCS 1826. Springer-Verlag, July 2000.

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

[TS97] W. Taha and T. Sheard. Multi-stage programming. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 321–321, June 1997.