Type-Based Publish/Subscribe*

Patrick Eugster † Rachid Guerraoui† Joe Sventek‡
†Swiss Federal Institute of Technology, Lausanne
‡Agilent Laboratories Scotland, Edinburgh

Abstract

This paper presents type-based publish/subscribe, a new variant of the publish/subscribe paradigm. Producers publish message objects on a communication bus, and consumers subscribe to the bus by specifying the types of the objects they are interested in. Message objects are considered as first class citizens and are classified by their types, instead of arbitrarily fixed topics.

By reusing the type scheme of the language to classify message objects, type-based publish/subscribe avoids any unnatural subscription scheme and provides for a seamless integration of a publish/subscribe middleware with the programming language. Type-based publish/subscribe has several quantifiable advantages over other publish/subscribe variants. In particular, the knowledge of the type of message objects enforces performance optimizations when combined with dynamic filters for content-based subscription.

Our type-based publish/subscribe prototype is based on Distributed Asynchronous Collections (DACs), programming abstractions for publish/subscribe interaction. They are implemented using GJ, an extended Java compiler adding genericity to the Java language, and enable the expression of safely typed distributed interaction without requiring any generation of typed proxies.

Keywords
Objects, distribution, asynchrony, publish/subscribe, typing, collection, messaging, genericity

1 Introduction

This paper introduces type-based publish/subscribe: a new variant of distributed messaging based on the publish/subscribe paradigm. Publish/subscribe interaction is a message-centric approach to communication in a distributed environment, characterized by the strong decoupling of participants in *time* as well as *space*.[1] Producers publish information on an information bus [OPSS93] and consumers subscribe to information they want to receive.

Classical approaches to publish/subscribe messaging involve different models for the middleware and the programming language. Consequently, the object-oriented and message-oriented worlds are often claimed to be incompatible [Koe99]. Our type-based publish/subscribe variant makes an attempt to reunite both worlds, by viewing messages as first class objects and letting consumers subscribe to such message objects by subscribing to the *types* of message objects they want to receive.

Figure 1 illustrates the intuitive idea underlying our approach. Participant P_1 subscribes to type A, which can be viewed as connecting to a flow of published message objects conforming to type A. Through subtyping, P_1 also receives objects of types B and C, and as a consequence, D. On the other hand, participant P_3 subscribes to objects of type D, while it publishes objects of types B and D.

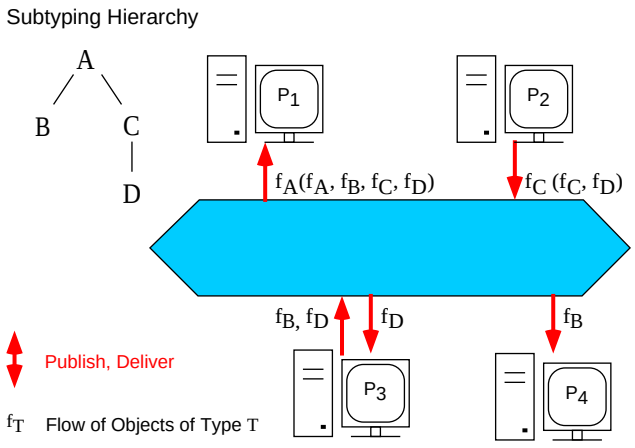


Figure 1. Type-Based Publish/Subscribe

*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.

Integrating programming language and middleware. With the increasing complexity and distribution of computing systems, separation of concerns [KLM+97, TOHS99] becomes important to isolate problems, but aspects which are not quite orthogonal are sometimes tackled independently. This issue has already been identified for instance in the domain of object-oriented data management systems [BZ87], where two separate languages coexist; one for the definition of data and another one for the manipulation of data with the management system. A similar mismatch has occurred between object-orientation and practices in message-oriented middleware. Publish/subscribe middleware systems introduce subscription schemes which are artificial and reduce messages to the data they carry.

The notion of type-based publish/subscribe we present in this paper represents a subscription scheme which is fully based on a natural classification of messages according to their type. It differs fundamentally from the notion of type-based subscription introduced in [HLS97], by fully relying on the type as an *inherent* attribute of objects and not on an *external* property added to objects in order to express a language-extrinsic subscription scheme.

Type-based publish/subscribe in perspective. Besides enabling a seamless integration of middleware and language, type-based publish/subscribe has several advantages over the traditional publish/subscribe styles, namely the *topic-based* (*subject-based*) style [TIB99, AEM99, Ske98], based on group-like notions [Pow96], and the *content-based* publish/subscribe variant [Car98, SA97, BCM+99], which enables applications to describe runtime properties of messages they wish to receive.

Common topic-based systems arrange topics in hierarchies and denote them with URL-like references. Such references can not express *multiple specialization*, i.e., a topic can not have several supertopics. Types offer a natural way of doing this, provided that the language offers multiple subtyping. The static nature of topics, which enables efficient implementations by reusing known multicast primitives, is often seen as its limitation. An increased *expressiveness* [Car98] is obtained with content-based publish/subscribe, in which subscribers describe runtime properties of the message objects they wish to receive. Such a subscription scheme tends to violate object encapsulation, and is usually realized with a subscription language. The expression of custom filters can be made with a filter library in order to avoid any subscription language and preserve encapsulation, but might lead to dynamic code, i.e., using *reflection* [KdRB91]. As we will illustrate in this paper, the knowledge *a priori* of the runtime type of messages in type-based publish/subscribe enables the generation of static filter code from dynamically implemented instances of a general filter library.

Generic DISTRIBUTED ASYNCHRONOUS COLLECTIONS. In classical middleware approaches based on derivatives of the *remote procedure call* (so-called *method-based* middleware, e.g., CORBA [OMG98a], DCOM [Mic99], or JAVA RMI [Sun99a]), precompilers are widely employed for the generation of typed proxies. In some standard approaches to typed event-based communication this technique is used as well [OMG98b], while other solutions force the application to provide its own proxies [Obe00], or even to make explicit type casts [HBS98, Sun99c, Sun99b]. In this paper we describe a way to express type-safe message-oriented interaction based on the publish/subscribe paradigm through *genericity* (*parametric polymorphism*). With our approach, explicit type casts in the application code can be avoided, and precompilation becomes obsolete. The realization presented in this paper combines the power of DISTRIBUTED ASYNCHRONOUS COLLECTIONS (DACs) [EGS00], abstractions which enable the expression of various publish/subscribe variants, with GJ [BOSW98], an extended JAVA compiler which offers generic types and methods.

Contributions. In short, this paper introduces our type-based variant of the publish/subscribe interaction model. We show how types enforce a natural and inherent subscription scheme, avoiding any explicit message classification through topics. We furthermore depict how parametric polymorphism can be used to express typed event-based distributed interaction, circumventing any type casts or generation of typed proxies. And finally, we illustrate how our generic DACs promote the efficient use of content-based filters with an underlying type-based scheme, in a way that preserves encapsulation and circumvents the need for any subscription language.

Roadmap. The remainder of this paper is organized as follows. Section 2 first recalls the topic-based and content-based publish/subscribe interaction styles and points out their limitations. Section 3 gives an overview of our notion of type-based publish/subscribe. We introduce a simple formalism to help specifying the semantics of "subscribing to a type" and to point out a peculiarity of the JAVA type system. Section 4 shows how we have used DACs to express type-based publish/subscribe. Section 5 presents details of our realization of generic DACs based on GJ, while Section 6 illustrates how type-based publish/subscribe provides for a highly flexible and efficient subscription mechanism when combined with content-based filters. In Section 7 we discuss some issues related to GJ and types in JAVA, and contrast our efforts with current practices in method-based middleware. Section 8 presents an overview of related approaches to typed event-based interaction. Finally, Section 9 concludes the paper.

2 Common Publish/Subscribe Models

The publish/subscribe paradigm is a loose communication scheme for modeling the interaction between applications in distributed systems. Unlike the classic *remote procedure call* model, publish/subscribe provides *decoupling* in time and in space. This makes publish/subscribe very attractive for large scale interaction. Nevertheless current publish/subscribe models, namely the *topic-based* and *content-based* interaction models, present both significant shortcomings.

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., when used for replication [Bir93]), topics typically overlap, i.e., a process participates (in the role of publisher and/or subscriber) in more than just one topic.

Referencing topics. In order to classify topics more easily, it is of great use to furthermore arrange them in a hierarchy [TIB99, AEM99, Cor99]. In this model, a topic can be a derived or more specialized topic of another one, and is therefore called *subtopic*.

Connecting to a topic thus requires the name in a URL-type format. Typically, if the *department of communication systems* (DSC) of the *Swiss Federal Institute of Technology* (EPFL) organizes talks, it could notify these to its collaborators through a topic */DSC/Talk*, as shown in Figure 2. */DSC/Talk* is a subtopic of */DSC*.

Subscribing to a topic can trigger subscriptions for the subtopics as well. In Figure 2, subscriber S_1 subscribes to topic */DSC* and claims its interest in all subtopics. Hence S_1 does not only receive message m_2 but also message m_1 published for topic */DSC/Talk*. In contrast, S_2 only subscribes to */DSC/Talk* and thus not only receives message m_2, which belongs to the *supertopic /DSC*.

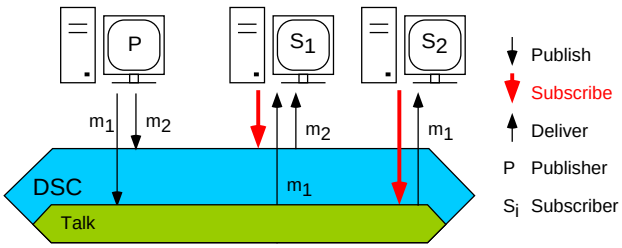


Figure 2. Topic-Based Publish/Subscribe

Limited expressiveness. The static classification of messages in topics makes efficient implementations possible. Multicast protocols have been largely studied in the context of group communication [BJ87], also focusing on scalability issues [KKT96]. Topics are thus often mapped to such multicast groups, which can be pictured as *dissemination channels*.[2] A widely employed primitive is *IP multicast* [Dee94]. IP multicast is inherently unreliable, but further multicast protocols offering reliability are emerging [SSLB97], and it seems straightforward to map topics to such multicast groups.

However, the division of the message space in *topics* can be seen as "artifical" and leads to a *coarse-grained* classification of messages. Consumers might subscribe to topics, but only be effectively interested in messages corresponding to strongly individual and finer criteria, which leads to an inefficient use of bandwidth. The same might occur when introducing a large number of topics to achieve a fine-grained taxonomy, since each topic might see the occurrence of only few messages. This presents a severe drawback for solutions based on IP multicast, since such addresses are limited in number. Disposing topics in hierarchies does not circumvent the problem, because subtopics are handled just like top-level topics, in the sense that they are mapped to separate channels: a message published for the topic referenced by */DSC/Talk*, will be disseminated in a channel */DSC/Talk*, and not in */DSC*. Consumers that have subscribed to the subtopics of */DSC* are automatically "connected" to */DSC/Talk*.

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 [Car98, SA97, BCM+99].[3] This new feature gives more expressiveness and flexibility to the application, by removing entirely the limitations of statically 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 classified according to *arbitrarily* fixed criteria, but by their runtime properties.

Subscription patterns and filters. Each subscriber only receives the notifications that match entirely its individual criteria. Such application requirements can be seen as a *pattern* against which messages are matched, and are translated to *filters*. Applying such filters enables the drastic reduction of unnecessary message transfers.

Figure 3 shows the difference to topic-based subscribing.

[2] The CORBA EVENT SERVICE [OMG98b] provides the application with *channels* at the programming abstraction level.

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

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 pattern 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 pattern of subscriber S_2, and is hence not delivered to subscriber S_1.

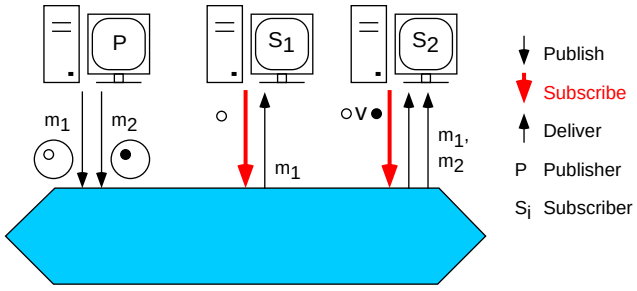


Figure 3. Content-Based Subscribing

The cost of dynamism. Content-based publish/subscribe removes limitations of the static topic-based flavor, but the dynamism it brings requires sacrifices.

Efficiency: Content-based publish/subscribe requires specifically tuned dissemination protocols. In fact, the matching of events against subscription patterns should take place as close to the source as possible, in order to avoid unnecessary message transfers. On the other hand, one should not end up in the case of matching a message at the publishers site against the pattern of each subscriber, since this would require every publishing participant to know every subscriber, which is usually unfeasible at large scale. Therefore the filtering has to take place *in transit*, which makes the reuse of known group multicast primitives and efficient realizations hard (e.g., [OAA+00]).

Encapsulation: There is another difficulty which we will devote more consideration in the context of this paper: describing properties of messages usually means introducing a subscription language, which makes it cumbersome to express complex patterns. Such subscription languages enable the description of message objects but are usually orthogonal to the programming language, in which those message objects are implemented. Furthermore they tend to violate encapsulation, by using the attributes of messages to describe requirements.

To avoid subscription languages, the application can be asked to provide filters in the form of executable code. These filters are however opaque to the middleware system and are not necessarily safe. Furthemore, avoiding redundant queries on message objects (like the ones proposed in [ASS+98]) becomes difficult.

Reflection-based filters There are indeed ways to circumvent these problems, for instance by using reflection [KdRB91] to implement libraries of filters. We have implemented such filter objects which permit the sorting of message objects by calling methods specified by the application on those objects and comparing the results to predefined values, in a way similar to approaches chosen in object-oriented data management systems, e.g., [SO95] (see Figure 4).

With reflection however, a reference to a method is obtained at runtime. This implies a *dynamic lookup*, which is very costly. Moreover, it is summed with the overhead of the *dynamic invocation* itself, which results in an important latency. Figure 5 shows the cost of such dynamic interaction by comparing the evaluation of basic filters encapsulating method calls with a varying number of arguments (between 0 and 9 JAVA objects) using (1) dynamic invocations with dynamic method lookup, (2) dynamic invocations without method lookup (assuming that all objects are of the same type, and thus the lookup is made once and for all), 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). Note that these results were obtained without any *Just In Time* (JIT) compiler. The speedup factor observed for static invocations when using a JIT compiler was over three (!), while the speedup in the case of dynamic evaluation was, as expected, insignificant.

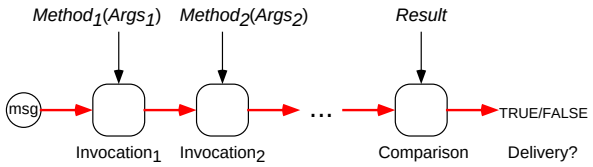


Figure 4. Content-Based Filtering With Reflection

The next sections show how a type-based scheme enforces a seamless integration of the publish/subscribe middleware with the language, and how performance can be improved when combining with content-based subscription facilities.

3 Type-Based Publish/Subscribe

This section introduces type-based publish/subscribe, a new approach to classifying messages in a *many-to-many* publish/subscribe interaction environment. The discussion is made in a general context, but is illustrated through the type system of the JAVA programming language. We will

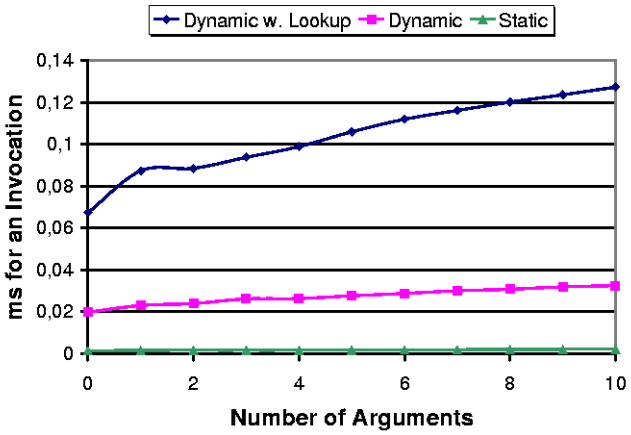


Figure 5. Cost of Dynamism in Content-Based Filtering

describe our concrete implementation of the idea in Sections 4 and 5.

3.1 Overview

Roughly spoken, our type-based publish/subscribe scheme consists in using the type scheme of an "ordinary" programming language without explicitly introducing a topic hierarchy, nor any other specific notion of message kind. A subscriber advertises its interest in a type *T*, which means that it will receive all messages of type *T*.

This idea has been mainly driven by the observation that in practice messages regrouped in a topic usually are of the same type because they are somehow related, i.e., they notify the same kind of event. This design choice furthermore enables the straightforward interpretation of messages belonging to a same topic, which is not obvious since most topic-based systems only offer weakly typed interfaces. Introducing a *key* in the form of a topic seems superfluous to regroup *message objects* that already express an affiliation through an implicit attribute – their type. This awkward situation can be avoided by simply matching the notion of *event kind* with that of an *event type*. Figure 6 shows a comparison of topics and types, based on the case of a type `Talk` used to advertise talks to collaborators of the DSC.[4]

3.2 Fundamentals

Type-based publish/subscribe is a static classification scheme based on types. Therebey we implicitly assume that the considered type system is static. In order to describe our notion of type-based subscribing more precisely, we establish a general model of a static type system which can be extended to capture different concrete languages. This will help us shed light on a specific characteristic of the JAVA type system with respect to type-based subscribing.

Subtyping. Just like topics can have subtopics, types can have subtypes, but subtyping is not handled the same way in every language. In strongly typed object-oriented languages like *C++* [ES92] or *Eiffel* [Mey92] for instance, the inheritance hierarchy determines the conformance (subtype) relation. In such type schemes, the notions of *type* (*abstract type*, *type definition*, *interface*, *signature*) and *class* (*concrete type*, *type implementation)* are identical. Thus, subscribing to a type means subscribing to all instances of the indistinguishable class and its inheriting subclasses.

As already claimed in [CHC90] however, inheritance and subtyping should be clearly separated. To achieve this, type definitions must be separated from type implementations. Several solutions have appeared to augment existing programming languages that lack this separation, e.g, by introducing signatures for C++ [BR97]. We continue this study with separation of these two aspects in mind. The idea is neither to make any statement on what features a modern programming language should have, nor to engage a language-wise discussion of the semantics of type-based publish/subscribe. A model based on separation of types is simply more general, in that it can be applied to certain languages lacking such separation.

A simple model. To ease our following discussions, we introduce a simple formalism. In our context, we are mainly interested in the relationships between *types* and *classes*. Therefore we model the environment Γ as a pair $(Types, Classes)$, without introducing variables.

The relation of *subtyping* is denoted by $\leq$. In other terms, if T_2 is a subtype of T_1, then $T_2 \leq T_1$, and $T \leq T$. The relation of *inheritance* is denoted $\subseteq$, i.e., if a class C_1 inherits from C_2, then $C_1 \subseteq C_2$. $\Gamma \vdash C \subseteq C$ means that C is declared in Γ. With $\leq$, the same can be expressed for types. A class C which *implements* a type T leads to the notation $C \sqsubseteq T$. $\sqsubseteq$ will later also be used for single methods. The rules are summarized in Figure 7, but require certain clarifications:

Implementation transitivity: Since subtyping is not driven by inheritance, we can have the situation where $T_1 \leq T_2$, $C_1 \sqsubseteq T_1$, and $C_2 \sqsubseteq T_2$, but $C_1 \not\sqsubseteq C_2$. One could also argue that a class only implements a type (and its supertypes) if it explicitly declares so, contradicting *IMT1* and *IMT2*. In our context, we will however assume implementation transitivity.

Multiple inheritance: Multiple inheritance is difficult to handle, and its usefulness is often questioned [SCC+93]. For languages lacking separation of type definition and

[4] Remember that *DSC* stands for the Department of Communication Systems of the Swiss Federal Institute of Technology (EPFL).

	"Traditional" Topic-Based Publish/Subscribe	Type-Based Publish/Subscribe

Generic Message g	`public class Message {`
	`public final String topic;`
	`public final Object content;`
	`...`
	`}`
		(none required)

Message m	`public class Talk {...}`	`public class Talk {...}`
Criterion	topic of m is */DSC/Talk*	m is of type `Talk`
Argument	`String topic = "/DSC/Talk"`	`Class tClass = Class.forName("Talk")`
Evaluation	`g.topic.equals("/DSC/Talk")`	`m instanceof Talk` `tClass.isInstance(m)`
Deliver	`m = g.content`	`m`

Figure 6. Topic-Based vs. Type-Based Publish/Subscribe

implementation, it can be used to achieve multiple subtyping. In our case, we use only multiple subtyping, but a rule for multiple inheritance can be added to capture languages such as C++.

The notion of *widening* is not used in this context. Instead, we use $\sqsubseteq$ as a base for the notion of *conformance*: the objects which are of type T are instances of classes which *conform* to T. The set formed by these classes is denoted $\Xi(T)$, and the instances of these classes are also said to *conform* to T. For the following illustration on the JAVA type system, it is important to mention that this set spans all classes which implement type T via transitivity, i.e., *IMT1* and/or *IMT2* (Ξ_S in Figure 7), or *directly* (Ξ_D). Directly means that there is no subtype T_0 of T ($T_0 \neq T$) which they implement.

Separating dissemination and subscription. In common class-based object-oriented programming languages, an object is an instance of *exactly one* class, but can conform to *more than one* type. A message object should be disseminated by a single channel, and thus classes are mapped to dissemination channels. The class relationship $\subseteq$ defines the set of channels and their hierarchy. The type hierarchy given by the subtype relationship $\leq$ of the language, on the other hand, is used for subscribing.

Conformance, here reflected by $\sqsubseteq$, is the glue between types and classes. It is the main concern of the type-based middleware, which takes care of mapping a subscription to a type T to the channels associated to the classes that correspond to T. In topic-based systems, the same hierarchy defines the granularity of both subscription and dissemination, and these two aspects are hence not separated.

Subscribing to types. Now we can precisely define the notion of *type-based subscribing*. By default, subscribing to a type T triggers subscriptions to all of T's subtypes. Accordingly, we define $subs(T)$ as the action of subscribing to type T, and $recv(C)$ as receiving all published instances (message objects) of class C.[5] Inspired by *CC,* we can delineate the semantics of a subscription to a type more formally:

Subscription

$$\Gamma \vdash subs(T) \Rightarrow \Gamma \vdash \forall\, C \in \Xi(T) : recv(C) \quad \textit{(S)}$$

As shown by certain topic-based systems, subscribing to a node in a hierarchy without receiving messages corresponding to child nodes can be very useful. We therefore additionally define a notion of subscribing to a type *without* subtypes, represented by an action $subt()$:

Subscription without Subtypes

$$\Gamma \vdash subt(T) \Rightarrow \Gamma \vdash \forall\, C \in \Xi_D(T) : recv(C) \quad \textit{(WS)}$$

Until now, our formalism does not include any information about how types, resp. classes are declared. The declaration semantics make the whole difference between distinct languages. We illustrate the case of JAVA.

3.3 Illustration with JAVA

We have chosen JAVA as our implementation language mostly because it is well adapted for distribution and because it enables the definition of types without implementation.

JAVA type system. The goal of this section is not to formally analyze the JAVA type system, as this has been done in [Sym97, DEK99]. The idea here is to show the relationship to Γ introduced before, and thus explain how we have matched type-based subscription to JAVA types. For that purpose, we consider the JAVA type system without *native types* (`byte`, `int`, etc.), since we are mainly interested in application-defined types.

[5] To simplify, we assume *reliable* channels: a published message object is received exactly once by every correct subscribers.

Subtyping Transitivity

$$\frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3} \quad \textit{(ST)}$$

Inheritance Transitivity

$$\frac{\Gamma \vdash C_1 \subseteq C_2 \quad \Gamma \vdash C_2 \subseteq C_3}{\Gamma \vdash C_1 \subseteq C_3} \quad \textit{(INT)}$$

Implementation Transitivity

$$\frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash C \sqsubseteq T_1}{\Gamma \vdash C \sqsubseteq T_2} \quad \textit{(IMT1)}$$

$$\frac{\Gamma \vdash C_1 \subseteq C_2 \quad \Gamma \vdash C_2 \sqsubseteq T}{\Gamma \vdash C_1 \sqsubseteq T} \quad \textit{(IMT2)}$$

Multiple Subtyping

$$\frac{\Gamma \vdash T \leq \{T_1,\ ...,\ T_n\}}{\forall\, i\ \in\ [1,n]\, \Gamma \vdash\ T \leq T_i} \quad \textit{(MS)}$$

Class Conformance

$$\Gamma \vdash\ \Xi(T)\ =\ \{C \mid C \sqsubseteq T\} \quad \textit{(CC)}$$

Conformance via Subtyping

$$\Gamma \vdash\ \Xi_S(T) = \{C \mid \exists T_0 \neq\ T : T_0 \leq T, C \sqsubseteq T_0\} \quad \textit{(SC)}$$

Direct Conformance

$$\Gamma \vdash \Xi_D(T) = \{C \mid C \sqsubseteq T \wedge\ \nexists\, T_0 \neq\ T :\ T_0 \leq T,\ C \sqsubseteq T_0\} \quad \textit{(DC)}$$

Figure 7. Subtyping, Inheritance and Conformance

We introduce an environment Γ_J characterized by the sets of types, as well as interfaces and classes defining those types, as we will depict in the following. All rules given for Γ apply to $\Gamma_J = (Types, Classes, Interfaces)$. Figure 8 shows a formal specification of type and class declarations in Γ_J:

Explicit declaration: A type can be explicitly declared by declaring an interface `I`. The type declared by `I` is denoted T_I (*ETD*). Furthermore, if `I` extends another interface $\texttt{I}_0$, then T_I is a subtype of T_{I_0}, but $T_I \neq T_{I_0}$ (*ESD*).

Implicit declaration: Defining a class `C` implicitly declares a type, noted T_C, and at the same time gives the class C which implements it (*ITD*). If `C` implements an interface `I`, then T_C is a subtype of T_I, and thus according to *IMT1*, $C \sqsubseteq T_I$. If `C` implements a single interface `I`, and does not provide any `public` method m that does not implement a corresponding method declared s in `I`, then `C` does not define a *new* type. In that case $T_C = T_I$, and `C` represents a *pure* class, as shown by *PCD*. In all other cases T_C is a new type.

JAVA enforces single inheritance, which implies that a class hence never inherits from more than one superclass.[6] Multiple subtyping in JAVA is illustrated by *ESD* and *ISD2* in that interfaces or classes can subtype several superinterfaces.

Subscribing to JAVA types. Like in common class-based object-oriented languages, a JAVA object is an instance of exactly one class. Every JAVA class is thus mapped to a dissemination channel, but since a JAVA class can implicitly define a new type, subscriptions must be possible to classes as well as to interfaces. We discuss here the semantics of subscribing to JAVA types, which are formally presented in Figure 9.

Subscribing to an interface `I`: This triggers the subscription to all objects that conform to T_I. Subscribing without subtypes translates to subscribing to all classes in $\Xi_D(T_I)$, which includes all pure classes for type T_I (*IWS*). A subscription with subtypes also involves non-pure classes for T_I, i.e., classes implementing T_I (1) through transitivity or (2) directly but by adding at least one new method (*IS*). These classes correspond to $\Xi_S(T_I)$.

Subscribing to a class `C`: If `C` is a pure class, then subscribing to `C` is equivalent to subscribing to the interface `I` implemented by `C` (*PCS*, resp. *PCWS*). This triggers subscriptions to all pure classes (if any) of `I`, and these classes do not extend `C`. If `C` is not a pure class, the subscription includes only `C` (*CS*), unless subtypes are desired, in which case every class which inherits from `C` is included (*CWS*).

In JAVA, a pure class can be identified as such thanks to the *introspection* capabilities of the language. Through the functionalities in `java.lang.Class`, one can learn about the interfaces implemented by a class, as well as the number of `public` methods of those interfaces and the class itself.

[6] Like Smalltalk [GR83], JAVA presents a singly-rooted class hierarchy: except the root class `Object`, every JAVA class subsclasses *exactly one* superclass. If no inherited class is specified, `Object` is assumed.

Explicit Type Declaration

$$\frac{\Gamma_J = \Gamma'_J; \texttt{interface I} \ldots; \Gamma''_J}{\Gamma_J \vdash T_I \leq T_I} \quad \textit{(ETD)}$$

Explicit Subtype Declaration

$$\frac{\Gamma_J = \Gamma'_J; \texttt{interface I extends } \texttt{I}_1, \ldots, \texttt{I}_n; \Gamma''_J}{\Gamma_J \vdash T_I \leq \{T_{I_1}, \ldots T_{I_n}\}} \quad \textit{(ESD)}$$

Implicit Type Declaration

$$\frac{\Gamma_J = \Gamma'_J; \texttt{class C} \ldots; \Gamma''_J}{\Gamma_J \vdash T_C \leq T_C \quad \Gamma_J \vdash C \subseteq C \quad \Gamma_J \vdash C \sqsubseteq T_C} \quad \textit{(ITD)}$$

Implicit Subtype Declaration

$$\frac{\Gamma_J = \Gamma'_J; \texttt{class C extends C}' \ldots; \Gamma''_J}{\Gamma_J \vdash T_C \leq T_{C'} \quad \Gamma_J \vdash C \subseteq C'} \quad \textit{(ISD1)}$$

$$\frac{\Gamma_J = \Gamma'_J; \texttt{class C} \ldots \texttt{implements } \texttt{I}_1, \ldots, \texttt{I}_n; \Gamma''_J}{\Gamma_J \vdash T_C \leq \{T_{I_1}, \ldots T_{I_n}\}} \quad \textit{(ISD2)}$$

Pure Class Declaration

$$\frac{\Gamma_J = \Gamma'_J; \texttt{class C implements I}\,\{M_{T_C}\}; \Gamma''_J \quad \forall m \in M_{T_C}\ \exists s \in S_{T_I}\ m \sqsubseteq s}{\Gamma_J \vdash T_C = T_I} \quad \textit{(PCD)}$$

Figure 8. Type Declarations in JAVA

Interface Subscription

$$\Gamma_J \vdash subs_J(\texttt{I}) \Rightarrow \Gamma_J \vdash subs(T_I) \quad \textit{(IS)}$$

Interface Subscription without Subtypes

$$\Gamma_J \vdash subt_J(\texttt{I}) \Rightarrow \Gamma_J \vdash subt(T_I) \quad \textit{(IWS)}$$

Class Subscription

$$\Gamma_J \vdash subs_J(\texttt{C}) \Rightarrow \Gamma_J \vdash subs(T_C) \quad \textit{(CS)}$$

Class Subscription without Subtypes

$$\Gamma_J \vdash subt_J(\texttt{C}) \Rightarrow \Gamma_J \vdash subt(T_C) \quad \textit{(CWS)}$$

Pure Class Subscription

$$\Gamma_J \vdash subs_J(\texttt{C}) \wedge \exists \texttt{I} \in Interfaces(\Gamma_J) \quad \textit{(PCS)}$$
$$T_I = T_C \Rightarrow \Gamma_J \vdash subs(T_I)$$

Pure Class Subscription without Subtypes

$$\Gamma_J \vdash subt_J(\texttt{C}) \wedge \exists \texttt{I} \in Interfaces(\Gamma_J) \quad \textit{(PCWS)}$$
$$T_I = T_C \Rightarrow \Gamma_J \vdash subt(T_I)$$

Figure 9. Subscribing to JAVA Types

4 Type-Based Publish/Subscribe with DACs

This section first recalls our notion of DISTRIBUTED ASYNCHRONOUS COLLECTIONS [EGS00], programming abstractions for publish/subscribe interaction. In a second step, the use of DACs to express type-based publish/subscribe in JAVA is illustrated.

4.1 Background: DACs

We recall briefly the main characteristics of DACs. More details can be found in [EGS00].

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, in a way similar to a shared memory. DACs differ fundamentally from the distributed collections described in [Obj99], by being asynchronous and essentially distributed, i.e., DACs can be seen as ubiquitous entities.[7] Participating processes act with a DAC through a local proxy, which is viewed as a local collection and hides the distribution of the DAC. DACs are not centralized on a single host, in order to guarantee their availability despite certain failures.

The asynchronous flavor of DACs. 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*. According to the terminology adopted in the *observer design pattern* [GHJV95], the DAC is the *subject* and its client is the *observer*.[8]

A client expresses its interest in objects of a DAC by registering a *callback* object, through which the client will be notified of objects "pushed" into the DAC. Such a callback object implements the `Notifiable` interface. Figure 10

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

[8] In the observer design pattern, the subject is usually itself a publisher, and as a consequence, the event types are known and typed interaction can take place. In contrast, our DACs represent a *generic* event dissemination service based on the publish/subscribe paradigm, i.e., DACs are neutral mediators between anonymous publishers and subscribers.

shows the variant for topic-based publish/subscribe.

```
public interface Notifiable {

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

Figure 10. Notifiable Interface

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

In the sense of publish/subscribe, inserting an object into a DAC (`add()`, Figure 12) comes to publishing that object for the topic represented by the DAC.[9] Every DAC can thus be viewed as a publish/subscribe engine of its own. Subscribing to a topic equals subscribing to the DAC representing that topic, and can be expressed by registering a callback object through a method call reflecting specific semantics (`contains()` and `containsAll()` methods with `Notifiable` argument). When combined with filters for content-based subscribing, an object implementing the `Condition` interface given in Figure 11 must be registered. Such an object can be either (1) implemented by the application or (2) an instance of one of the various general purpose filters we supply. These are similar to the *predicate* objects used in [Obj99].

```
public interface Condition {

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

Figure 11. Condition Interface

4.2 Representing Types with DACs

Just as topics are represented by DACs for specific topics, type-based publish/subscribe can be expressed by DACs for specific types.

[9]The application is responsible for providing *serializable* objects in the sense of JAVA .

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

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

Figure 12. Interface DAC (Excerpt)

Subscribing with typed DACs. Subscribing to a collection of events which are of a given type T, implicitly means that the events of interest are those which conform to type T. By registering a callback object, the application can be notified of the occurrence of events that conform to type *T*. The DAC triggers subscriptions to all channels corresponding to the classes in Ξ, resp. Ξ_D.

In the case of JAVA, as explained in the previous section, a DAC must be able to represent a `class` or an `interface`. A JAVA type can be *reified* by an object, thanks to the reflective properties inherent to the language. A type is represented by an object of class `java.lang.Class`, and will thus be used as the key to distinguish types. In a simplified way, the underlying system can be pictured as maintaining a hierarchy of channels and their associated class objects. The DAC multiplexes the subscription to all channels representing classes which conform to T.

Publishing with typed DACs. In the same way, the application might publish objects of *any* type that conforms to T through a DAC that represents type T. Before a published object can be effectively disseminated by a DAC, the channel representing the class of the object must be determined first.

The determination of the precise channel also has to take place in the context of topic-based publishing, and we will show in 6, that JAVA offers the possibility of implementing this more efficiently with type-based publish/subscribe.

Strong typing with DACs. With type-based publish/subscribe, the message objects which are related and form a sort of topic conform to a same type. Compile-time type checking is therefore possible and should be en-

forced to reduce the number of explicit type checks and casts, and potential runtime errors. As a consequence, the interface offered by DACs for a given JAVA type T should offer methods where parameters representing message objects are of type T. The next section focuses on how to obtain such typed interaction in a very convenient manner by using *genericity*.

5 Pizza Delivery

A DAC representing a type offers an interface relying on that type. Generating typed interfaces and classes for every type is not the most convenient way of achieving this. The approach we have chosen with DACs is based on *genericity*, also known as *parametric polymorphism*.

5.1 Genericity

Our approach to realizing type-based publish/subscribe relies on genericity. In fact, a typed DAC for type T can be viewed as an instance of a *generic* class with a *type parameter* instantiated with T.

Such *generic* classes or types are integrated in certain languages like C++ (`template`, [ES92]) or Ada (`generic`, [Ada95]), while languages like JAVA or Oberon [Rei91] are designed to support subtypes directly, and to support generics by the idiom of replacing variable types by the top of the type hierarchy. For languages lacking such generic types and methods, adequate extensions have been widely studied. In the case of JAVA, several solutions have appeared like PIZZA [OW97], its follow-up GJ [BOSW98], NEXTGEN [CS98] and others [AFM97, MBL97, TT99].

Interestingly, the usefulness of such parametric types has been often demonstrated on collections.[10] It seems thus very promising to apply genericity to our DISTRIBUTED ASYNCHRONOUS COLLECTIONS as well.

5.2 Typed DACs with GJ

We have based our realization of type-based publish/subscribe on an existing solution, namely GJ. GJ has been chosen for several reasons: it is freely available, mature, has reaped much recognization from SUN, and enables the reuse of non-generic legacy code (*retrofitting*).

How GJ works. GJ enables the use of the original JAVA virtual machine, and comes as an extended JAVA compiler, yet fully compatible with the SUN release.

[10] Even in the case of C++, a good proof of the advantages of parametric types is given by the widely used *STL* collection framework [SL95].

Roughly spoken, GJ translates generic constructs to non-generic ones. GJ proceeds by erasing all type parameters, mapping type variables to their bounds, and inserting casts into the compiled class code. No specific classes or interfaces are thus created, as in the case of [CS98]. Type conversions do not take place in a generically defined class, but in the objects which invoke instances of that class, as schematically outlined in Figure 13.

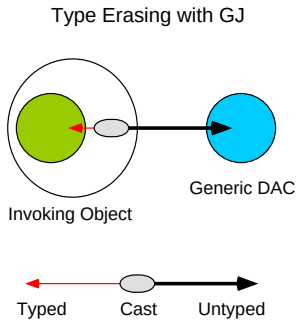


Figure 13. Interaction with a Generic DAC

Generic DACs. GJ presents certain limitations, which are detailed in [BOSW98]. For our application purpose however, these were not decisive. An immediate consequence of the use of GJ is that the generic `GDAC` interface can not subtype the `java.util.Collection` interface anymore, as visible in Figure 14. This is due to conflicts between original methods and the type-erased generic variants (e.g., `add()`).

Note that the `notify()` method of the generic `GNotifiable` callback interface does not require two parameters anymore. The same remark can be made for the generic `GCondition` interface in Figure 14. In the context of topic-based publish/subscribe, the second argument is used to denote the topic to which the message belongs, since the topic is viewed as an external property of the message object. In the case of type-based publish/subscribe however, the key is given implicitly by the type of the message object.

5.3 Programming with Generic DACs

We illustrate the convenience of distributed programming with generic DACs through the simple talk notification example introduced in Section 2, which we will extend in order to show further benefits of our type-based approach over a topic-based schemes.

The scenario. The Swiss Federal Institute of Technology (EPFL) also has a *computer science department* (DI). Like the DSC, the DI sometimes organizes talks, and talks held in either one of the departments are usually interesting for

```
public interface GNotifiable<T> {

  public void notify(T msg);
}


public interface GCondition<T> {

  public boolean conforms(T msg);
}


public interface GDAC<T>

{
  public T get();
  public boolean contains(T message);
  public boolean add(T message);
  ...
  public boolean contains(GNotifiable<T> n);
  public boolean containsAll(GNotifiable<T> n);
  ...
  public boolean contains(GNotifiable<T> n,
                          GCondition<T> c);
  public boolean containsAll(GNotifiable<T> n,
                             GCondition<T> c);
  ...
  public void clear(GNotifiable<T> n);
  ...
}
```

Figure 14. Generic Interfaces

researchers of both departments. Instead of thus having a separate topic to advertise upcoming talks of each department (*/DSC/Talk* and */DI/Talk*), forcing collaborators to subscribe to all, one could imagine having a single topic. Should it be subtopic of */DSC* or */DI*?

This situation can not be expressed conveniently with common topic-based naming schemes. With an extended naming scheme, the topic advertising talks of both DI and DSC could be denoted *(()DSC, ()DI)Talk*. Such a notation enables the resolution of the supertopics, but gives a somewhat cryptic representation of relationships.

As illustrated by Figure 15, type-based publish/subscribe allows the expression of such multiple specialization in a natural and convenient manner provided that the language enforces multiple subtyping.

The application. Figure 15 also shows the corresponding callback class that just prints the content of the received notifications. Figure 16 shows how to subscribe to talk notifications and to advertise new talks.

The constructor of the `GDASet` class requires an argument, which denotes the type (either by name or by `Class` object), with which the DAC will be used. Although it seems redundant to the type parameter, it is necessary. Since types are erased with GJ, there is no possibility for a generically defined class to determine at runtime for what type it is effectively being used, as long as it does not get hold of an instance. In the case of a generic collection, the type of the contained elements would only be learnt once an element is pushed into the collection. The DAC proxy used by a pure subscriber would have no way of determining to what dissemination channel(s) it must "connect", since no element would ever be inserted locally. The inconvenience of the constructor argument can thus be seen as the *cost incurred due to distribution*.

We are currently exploring the feasibility of our approach with alternative generic JAVA compilers. The approach chosen in [SA98] for instance provides a generic class with information about its type parameters at runtime.

```
public interface DSC {...}

public interface DI {...}

public class Talk
  implements java.io.Serializable, DSC, DI

{
  private String speaker;
  private String descr;
  public String getSpeaker() { return speaker; }
  public String getDescr() { return descr; }
  public Talk(String speaker, String descr) {
    this.speaker = speaker; this.descr = descr; }
}


public class TalkNotifiable
  implements GNotifiable<Talk>

{
  public void notify(Talk msg)
    { System.out.println(msg.getSpeaker());
      System.out.println(msg.getDescr()); }
}
```

Figure 15. Application Message and Callback Classes

```
GDAC<Talk> talkDAC = new GDASet<Talk>("Talk");
talkDAC.contains(new TalkNotifiable());
...
Talk myTalk = new Talk("Patrick Eugster",
                       "Type-Based Pub/Sub");
talkDAC.add(myTalk);
...
```

Figure 16. Type-Based Publish/Subscribe with Generic DACs

6 Evaluation

This section evaluates the benefits of type-based publish/subscribe over its direct rivals. The goal is not to give a quantitative performance evaluation of DACs, but to give qualitative values, showing the benefit of our type-based publish/subscribe, especially in combination with content-based filters. For that purpose, we compare between DACs for type-based and topic-based [EGS00] publish/subscribe respectively.

6.1 Testbed

Our measurements were made with the JVM 1.2, enabled JIT and native threads. We have chosen the same classification of messages for all measurements, first realized by explicitly defining a topic, and second by using the type of the messages as an implicit classification.

Messages. The message objects transmitted in our example scenario were of a specifically created class encapsulating an integer attribute. In the tests using content-based filters, the value v of the integer attribute was chosen arbitrarily out of a set $v \in [1, n]$, where every value had the same probability $p_v = 1/n$ of being chosen.

Network. Three interconnected local networks were chosen for the measurements. A single publisher was notifying events from a first network (SUN ULTRA 60, SOLARIS 2.6, 256 Mb RAM, 9 Gb harddisk) to subscribers equally distributed over two remaining networks; the first one consisting of all together 60 SUN SUPERSPARC 20 stations (model 502: 2 CPU, 64 Mb RAM, 1Gb harddisk and SOLARIS 2.6), and the second one consisting of 60 SUN ULTRA 10 (SOLARIS 2.6, 256 Mb RAM, 9 Gb harddisk) stations. The individual stations and the different networks where communicating via FAST ETHERNET.

6.2 Topic-Based vs Type-Based Publish/Subscribe with DACs

In a first step we compare the performances of *pure* type-based and topic-based publish/subscribe. The difference is little, and slightly in favor of type-based publish/subscribe.

Preliminary: type inclusion vs topic inclusion. Checking an object for its conformance to a type is a well-studied problem in object-oriented languages and can be realized very efficiently, as conveyed by [VHK97]. This observation is very much in favor of using the type system of the language to classify messages, instead of adding attributes to message objects.

In JAVA, checking an object for conformance to a type appears to be less costly than querying one of the object's attributes through a method call and comparing it to a given value.[11] Figure 17 compares the latency of classifying message objects by evaluating their topic with the latency resulting from checking the same message objects for their conformance to a type. These tests have been made on a SUN ULTRA 60 (SOLARIS 2.6, 256 Mb RAM, 9 Gb harddisk) with JAVA 1.2 (native threads).

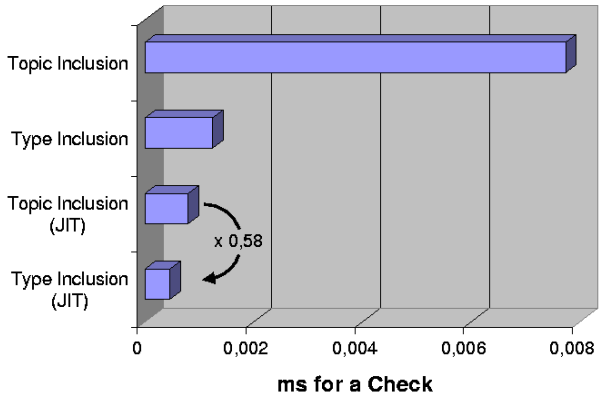


Figure 17. Checking the Affiliation of a Message Object

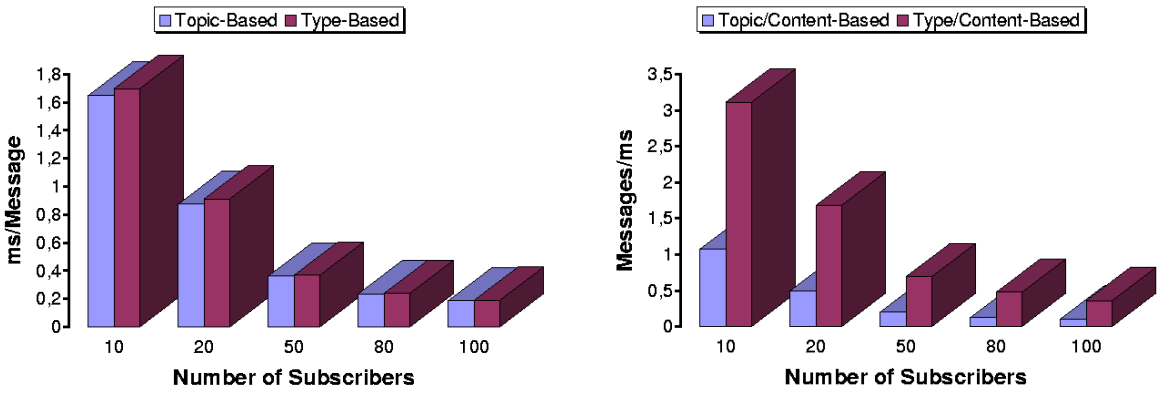
Casts and channel lookups. The casts that are inserted by GJ do not penalize generic DACs. In fact, with topic-based publish/subscribe based on DACs, the casts must be made as well, but are explicitly inserted in the client code by the application developer (e.g., in the `Notifiable` object).

Figure 18(a) compares the publishing of messages with type-based and topic-based publish/subscribe. The difference in throughput results from the faster classification of published messages thanks to efficient type inclusion tests. This difference is independent of the number of subscribers, since the channel lookup is done exactly once for every published message. The difference in terms of throughput decreases thus with an increasing number of subscribers. Note however that the scenario shown in the figure is based on a single channel (corresponding to a single topic, resp. message type), and thus the lookup was very fast. With an increasing number of channels, the difference in terms of latency becomes more important.

6.3 Adding Filters

As previouly explained, static schemes like the one introduced with type-based publish/subscribe can be implemented efficiently, but offer little expressiveness. On

[11] `instanceof` operator vs `startsWith()` method in `java.lang.String`.



(a) Topic-Based vs Type-Based Publishing

(b) Throughput with Matching Rate of 50%

Figure 18. Cost of Publishing Messages and Cost of Content-Based Matching

the other hand, the content-based approach enables finer grained subscription criteria, but implementations suffer from the cost claimed by the introduced dynamicity. It is hence not possible to say which style is more convenient. Our DACs combine the two, by enabling the application of a content-based filter to a (sub)hierarchy of topics or types.

A simple example. In our running example on talk notifications, some researchers might only be interested in talks concerning certain subjects, like "Java" or "Multicast", or given by certain speakers.

Figure 19 shows how to restrict the talks to the ones given by a specific person. The filter class `Equals` used here compares the return value of a method with a predefined value in the sense of the JAVA `equals()` method. The second constructor argument corresponds to the parameter list for the method call, which is empty in this case.

```
GDAC<Talk> talkDAC = new GDASet<Talk>("Talk");
Condition myEquals =
  new Equals("getSpeaker", null,
             "Patrick Eugster");
Notifiable<Talk> notifiable =
  new TalkNotifiable();
talkDAC.containsAll(notifiable, myEquals);
```

Figure 19. Defining a Simple Condition

Filters and dynamism. The filters we provide as a library, like the `Equals` class, cover a broad spectrum of possible application requirements, and programmers can extend these by following certain guidelines. We have initially designed our filter objects for general use, i.e., pure content-based publish/subscribe. The application can define a method to filter message objects either (1) directly through the corresponding method object (class `java.lang.reflect.Method`), or (2) by the method name and signature, as shown in Figure 19. In the first case, the type of the message objects is known, since in JAVA a `Method` object is bound to a type. This already gives the possibility to generate static code, and to avoid costly dynamic method invocations.

However, for most application programmers reflection is not straightforward to use, and it is much easier to initialize a filter with a method name and the desired arguments, as shown in Figure 19. In this case, the type of the message objects is unknown, unless the filter is applied to a generic DAC for type T, in which case the type of the message objects, T, is given implicitly. This enables the entire circumvention of any dynamic code by generating and compiling static filter code at runtime. Figure 20 outlines how the condition `myEquals` in Figure 19 can be used to generate a static condition. As we will show in the following, the speedup increases significantly.

We have also applied runtime compilation to enforce static type checks in JAVA (`instanceof` operator instead of the more costly dynamic `isInstance()` counterpart in `java.lang.Class`) for the determination of the dissemination channel. The cost of dynamic compilation can usually be neglected, since subscriptions/unsubscriptions are rare compared to the high number of application messages.

Matching rate of 50% Figure 18(b) compares the respective throughputs of (1) topic-based publish/subscribe combined with dynamic filters and (2) type-based publish/subscribe with static filters generated at runtime. In this case, filters evaluated true in 50% of the cases (the value of the integer carried by a message was arbitrarily chosen between 1 and 2). The difference in terms of latency is no

	Dynamic Filter	Resulting Static Filter
Message Object	`Object o`	`Talk t`
Required Result	`String rResult = "Patrick Eugster"`	`String rResult = "Patrick Eugster"`
Method Arguments	`null`	(none)
Method	`Class c = o.getClass()` `Method m = c.getMethod("getSpeaker", null)`	("getSpeaker")
Invocation	`Object result = m.invoke(o, null)`	`Object result = t.getSpeaker()`
Delivery Decision	`result.equals(rResult)`	`result.equals(rResult)`

Figure 20. Dynamic and Static Filters

longer constant but increases significantly with the number of subscribers. This reflects in the throughput, which is roughly three times higher in the case of static filters with type-based publish/subscribe.

Matching rate of 25% The same tests were made with a satisfaction of the subscriber requirements for 25% of the messages and are reported in Figure 21(a). In this case the speedup factor reaches roughly four.

Dependency. Finally, Figure 21(b) shows the dependency between the matching rate of subscriber criteria and the benefit of type-based publish/subscribe, by varying the number of different possible values carried by the messages and keeping the number of subscribers constant (100). With a decreasing matching rate, the advantage of using type-based publish/subscribe becomes visible. This is due to the fact that matching is actually very expensive, also compared to the effective sending of messages (in our case datagrams over UDP sockets). On the other hand, with high matching rates, the use of a finer static classification without filters can bring even better results.

Throughput decreases rather linearly with an increasing number of subscribers. Scalability can be improved by using intermediate *repeaters*.[12] In our case, we have omitted them to emphasize differences between different publish/subscribe styles with less than 1000 subscribers.

7 Discussion

This section discusses several issues related to generic DACs. Limitations of generic DACs are discussing focusing on distribution and the JAVA language and type system.

7.1 Subscribing to Nested JAVA Types

JAVA supports nested types,[13] which means that a type can be declared within another. There are four different ways of declaring nested types in JAVA. We discuss here which nested types can be used to parameterize DACs.

Anonymous Classes: These are classes that have no name. They combine the syntax for class definition with the syntax for object instantiation. Such classes cannot be referenced and can thus not be used with GJ, i.e., generic DACs can not be parameterized with anonymous classes.

Local Classes: A local class is defined within a block of JAVA code, and is visible only within that block. It can therefore not be defined as `public`. Consequently, generic DACs can not be parameterized with local classes.

Static Member Classes/Interfaces: A class (or interface) of this kind behaves much like a top-level class (or interface), except that it is declared within another class or interface. A static member class can access `static` fields and methods of the containing class. Such classes can be used in conjunction with generic DACs, as long as they are `public` and of course serializable.

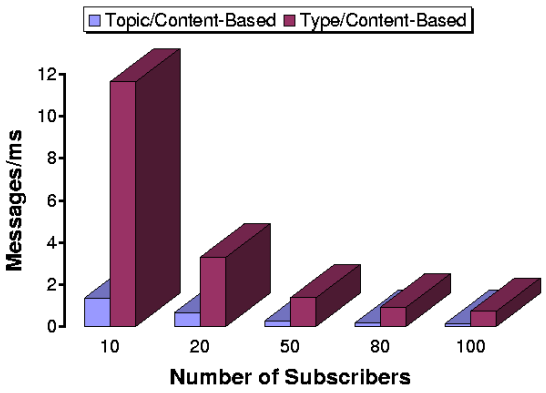
Member Classes: This last kind of nested class is much like the previous one, except that an instance is associated with an instance of the class in which the member class is declared. All fields and methods of that associated instance are thus accessible as well. DACs can be parameterized with such member classes, provided that these are `public` and serializable. Note that the containing class must be serializable as well, since the associated instance of the containing class must be transferred with the member class instance in order to always be accessible to the latter one.

7.2 Exploiting Multiple Subtyping

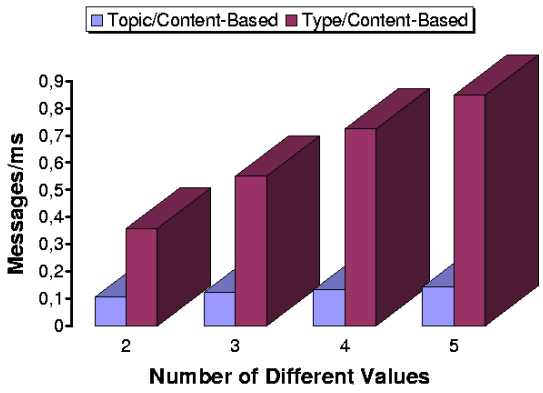
As explained in [BW98], JAVA uses *name equivalence* of types, which means that two types are compatible only if declared so. If two different types have the same parents in their type hierarchy, instances of one type can not be assigned to variables of the other's type. A type scheme which allows this is said to enforce *structural equivalence* of types. Both equivalences have their advantages and drawbacks,

[12] These are also called *event servers* [Car98], *routing daemons* [TIB99] or *message brokers* [ASS+98].

[13] According to [Mad99], we will avoid the term *inner class*.



(a) Matching Rate of 25%



(b) Summary

Figure 21. Throughput with Content-Based Matching (cont'd)

and the issue here is not to discuss which one is better than the other. Nevertheless, it would be interesting to restrict a subscription to classes which implement more than one type $T_1, ..., T_n$), without having to explicitly introduce a new type. According to *MS*,

$$\Gamma \vdash subs_p(T_1, ..., T_n) \Rightarrow$$
$$\Gamma \vdash \forall T \leq \{T_1, ..., T_n\} : subs_p(T)$$

In that case, the subscriber only receives a subset of the messages it would receive when subscribing to the subtypes of only one of the T_i.

Figure 22 shows an example expressed with the syntax of [BW98]. Instead of subscribing to several classes of talk notifications, the application can delineate more easily that it is interested in all talks of interest for both DI and DSC only (whether the *deparment of electricity* (DE) or others are also addressed or not). [BW98] is based on a specialized compiler, contradicting the use of GJ. [LBR96] is another approach based on a modified JVM.

7.3 Using Topics for Types

Most industrial strength solutions currently provide a form of topic-based publish/subscribe. We show how the strong resemblance between topics and types can be exploited for the realization of a type-based publish/subscribe system by reusing (parts of) an architecture developed for a topic-based one. With JAVA, the class *inheritance* hierarchy can be straightforwardly mapped to a topic *name* hierarchy. A class `DSC` can be mapped to a topic */DSC*, and if a class `Talk` extends `DSC`, then `Talk` can be mapped to a subtopic */DSC/Talk*. This mapping is possible, thanks to the two following properties of JAVA :

```
public interface DSC {...}

public interface DI {...}

public interface DE {...}
...

abstract interface DIandDSC extends DSC, DI;
...
GDAC<DIandDSC> talkDAC =
  new GDAC<DIandDSC>("DIandDSC");
DIandDSC t = talkDAC.get();
...
```

Figure 22. Exploiting Multiple Subtyping for Precise Subscribing

Unique Class Names: The naming conventions of JAVA stipulate that class names are unique [GJS96]. This gives an even stronger guarantee than actually required to ensure that no two distinct classes can be mapped to the same topic. In fact, it is sufficient if the URL-based name obtained by following the inheritance path of a class is unique. In other terms, two classes could very well have the same name, but would have to produce at least one different class in their inheritance path. Otherwise two participants can introduce incompatible homonymous types, leading to runtime errors.

Single Inheritance: In fact, with a traditional topic scheme, a topic is subtopic of *at most one* supertopic. A single inheritance scheme can be mapped in a straightforward way to such a topic hierarchy, but the expressing of multiple inheritance requires some modifications to a URL-like naming scheme (cf. Section 5).

7.4 Generic DACs in Perspective

Typed distributed interaction, as promoted by our generic DACs, is also encountered in current middleware solutions based on derivatives of the remote procedure call, where an interface description is precompiled in order to generate typed *stubs* and *skeletons*. We compare here our generic DACs with such method-based middleware, whereas the next section compares our generic DACs to typed event-based middleware.

Separating compilation. As an example, CORBA [OMG98a] relies on an *IDL compiler*, which is used to map IDL descriptions to language-specific constructs. We have first pursued a similar approach by generating typed DAC interfaces and classes with a specific precompiler for every message type T used by an application.

The development of an application based on this approach must however follow an exact order. After completing the message class, the typed DAC classes and interface can be generated. Typed DAC classes can best be pictured as typed wrappers, which invoke untyped constructs underneath. They mainly take care of type casts, as outlined in Figure 23. Finally, the application code can be compiled.

With CORBA, the precompilation phase is especially required because interfaces are described in a specific language. With JAVA RMI [Sun99a] typed stubs and skeletons are generated through the `rmic` compiler beforehand, although interfaces are described in JAVA. The application does not explicitly deal with its specific stubs and skeletons, and the separated compilation, first for remote interfaces and then for the application code, is not a necessity but promotes *separation of concerns*. In CORBA, separation aims primarily at enforcing interoperability.

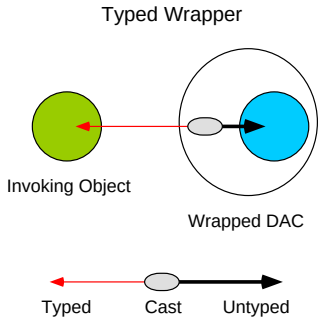


Figure 23. Generating Typed DACs

Stubs, skeletons and DACs. With a remote method invocation à la CORBA, stubs and skeletons are mainly required to transparently marshal and unmarshal requests, respectively. The former are located on the client side, and offer an interface that corresponds to the remote server object. In the general case, a server object is bound to a single skeleton, but every object bearing a reference to a server object has its own local stub.

DACs act both as stubs and as skeletons as shown in Figure 24. In fact, this symmetric proxy model is more adequate in our context since the DAC is not bound to a single consumer or producer. In the case of push-style interaction, a consumer implements a `Notifiable` interface, through which it is invoked. In the case of generic DACs, typed interfaces and classes do not really exist. Casts are inserted in the client code by GJ, and generic DACs can thus be viewed as *virtually typed proxies*.

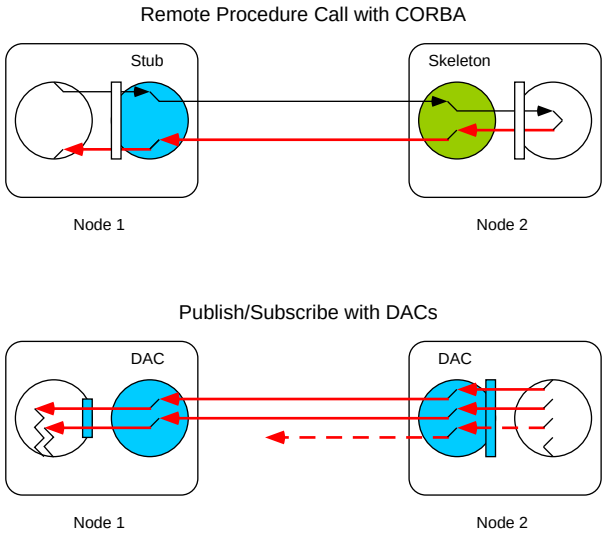


Figure 24. DACs vs. Stubs/Skeletons

8 Related Work

Recently, 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 efforts. We present here the main characteristics of specifications which make use of typed events, by giving the application substantial freedom in defining its own event types, instead of using message types defined as part of the messaging API.

CORBA EVENT SERVICE. The OMG has specified a CORBA service for publish/subscribe oriented communication, known as the CORBA EVENT SERVICE [OMG98b]. 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, a consumer interacts with an *event channel* expressing thereby an interest in receiving *all* the events from the channel. In other words, a filtering of events is done according to the channel names, which basically correspond to topic names. Furthermore, typed events can be used. The application can provide a specific interface, which defines its own operations through which it will be called, either to pull information or to be pushed. Because the event service supports one-way interaction, all operation parameters in the former case must be tagged `out`, and in the latter case `in`. Typed proxies are generated based on the application's interface, which in practice requires a specific compiler.[14] For many users the specification for typed interaction is difficult to understand, and implementors find it hard to deal with [OMG00]. Most implementations therefore only provide untyped events.

CORBA NOTIFICATION SERVICE. The deficiencies of the CORBA EVENT SERVICE, such as the difficulties with typed events described above as well as missing support for QoS and realtime requirements, were apparent soon after commercial implementations became available [SV97]. After the emergence of extended and proprietary approaches aimed at fixing the shortcomings of the event service (e.g., [HLS97]), the OMG has issued a request for proposal for an augmented specification, the CORBA NOTIFICATION SERVICE [OMG98c]. A *notification channel* is an event channel with additional functionalities. Notions like priority and reliability are explicitly dealt with, and a new form of typed events, called *structured events* is introduced. Structured events can be seen as semi-typed events. They provide a general type of messages, which manifest attributes like event type and event name. They are roughly composed of an event header and an event body. Both parts consist of a fixed part (a fixed header, resp. a body carrying an `any`), and a variable part. The variable parts are structured as name-value pairs, to which applications map their specific needs. These structured events however only partly solve the problem of typing, because the specification only mentions a set of standardized and domain-specific mappings, while most applications will require their own mappings. Furthermore, in the context of content-based filtering, the name-value pairs are seen as the attributes of the message and are directly accessed through *filter objects*. Constraints are described as strings following a complex subscription grammar called DEFAULT FILTER CONSTRAINT LANGUAGE, which extends the OMG's TRADER CONSTRAINT LANGUAGE.

[14] COM+ EVENTS [Obe00] supports the same typed model, but only for push-style interaction. `EventClass` objects are a form of typed proxies, which must be provided by the application.

JAVA MESSAGE SERVICE. The JAVA MESSAGE SERVICE (JMS) [HBS98] is a specification from SUN. Its goal is to offer a unified JAVA API around common messaging engines. Commonalities and differences of topic- and.content-based publish/subscribe have been realized and integrated. In contrast to the CORBA EVENT and NOTIFICATION SERVICES, message queuing (*point-to-point* interaction) is supported as integral part of JMS. Nevertheless, since it aims at wrapping existing solutions, the support for content-based subscription is done in a way similar to the scheme encountered with the CORBA NOTIFICATION SERVICE. Properties are explicitly attached to messages, and are in practice faithful copies of the message attributes. Subscription patterns (*message selectors*) based on these properties, are expressed through strings following a specific grammar. The JMS API defines five types of messages with different payloads: text, bytes, stream, map and object messages. The last one can be used to transport any serializable JAVA object, but type casts are left to the application.

JINI DISTRIBUTED EVENT SPECIFICATION. The JINI DISTRIBUTED EVENT SPECIFICATION [AOS+99] is straightforwardly based on the observer design pattern. Registration of a `RemoteEventListener` with an *event generator* indicates the kind of events that is of interest, while a notification indicates an occurrence of that kind of event. Such notifications are instances of a type `RemoteEvent` [Sun99c], which is subtyped to express specific events. Unlike with generic DACs, the interpretation of the concrete event type is thus left to the application, including type casts.

JAVASPACE JAVASPACE is a more general approach [Sun99b, FHA99] to publish/subscribe notification. Inspired by LINDA's TUPLE SPACE [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. With a JAVASPACE one can thus build a publish/subscribe communication scheme in which the JAVASPACE plays the role of the event channel aimed at broadcasting event notifications to a set of subscriber objects. Custom events can be generated by subtyping the `Event` type [Sun99d]. To specify the type of events a subscriber S is interested in, S must always provide a template object o_t when subscribing to a JAVASPACE. A necessary condition for o, an object notifying an event, to be delivered to S is that o conforms to the type of o_t. Furthermore, the attributes of o and o_t are compared byte-wise, and `null`

plays the role of wildcard. Type checking is hence not enforced at compilation and explicit type casts are necessary inside the application code.

The specifications which offer type-safe interaction require explicit proxy generation. Furthermore, these approaches use the types of the consumers as classification scheme, and events are not reified. In other terms, the event type is seen as a composition of both the event information type and the event consumer type. Other approaches force the application to explicitly apply type casts on received events. In contrast, generic DACs use an event classification scheme purely based on the types of the effective event notifications; the message objects. By relying on parametric polymorphism, generic DACs moreover make type casts obsolete and circumvent any middleware-specific precompilation. Nevetheless, it would be interesting to see how one could implement services that comply with the above-described specifications using generic DACs.

9 Conclusions

The two paradigms of object-orientation and messaging have often been presented as contradictory [Koe99]. This is due to the way these aspects are handled in current middleware. With so-called *method-based* (also called *object-oriented*) middleware like CORBA [OMG98a] or JAVA RMI [Sun99a], remote objects interact via remote method invocation and distribution is often viewed as an implementation issue. 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.

As argued in [WWWK94, Lea97, Gue99] however, distribution transparency is a myth that is both misleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction.

With message-oriented middleware based on the publish/subscribe paradigm, the application becomes aware of distribution again, but the ugly aspects are often hidden behind abstractions like *topics* or *channels*. Such abstractions lead to classification schemes which are static and offer only little expressiveness. The content-based publish/subscribe style removes certain limitations of topic-based publish/subscribe, but presents other shortcomings: it is not only difficult to implement efficiently, but is also hard to express without violating encapsulation and without using a complicated subscription language (e.g., [Car98, SA97, BCM+99, OMG00, HBS98]).

Our type-based publish/subscribe variant marries the two worlds of object- and message-orientation, by considering messages as first class objects. In this way, one can avoid artificial classification schemes, e.g. based on topics. Message objects are classified according to their type and no additional property is attached to them. Consumers specify their interests by subscribing to types of message objects they wish to receive. This subscription scheme provides for a seamless integration of the programming language and the middleware platform. When combined with content-based filtering furthermore, the knowledge of the type enables the generation of static filters from dynamic application requirements expressed with filter objects from a library. Such an approach circumvents any unwieldy subscription language, preserves encapsulation, and can be implemented very efficiently.

Our implementation of type-based publish/subscribe combines two recent developments: it builds upon our work on DISTRIBUTED ASYNCHRONOUS COLLECTIONS (DACs) [EGS00], abstractions which alleviate any artificial distinction between different publish/subscribe styles, and merges it with the latest advancements on genericity in the Java language [BOSW98]. We have implemented generic DACs both on UDP and on top of the VISIBROKER CORBA EVENT SERVICE [Vis98].

The language characteristics we have chosen as a discussion base are very much driven by our implementation choice – JAVA. We are currently working on a more general analysis of the semantics of "subscribing to a type" with the goal to illustrate key concepts in different programming languages. We intend to reuse these results to enforce language interoperability by defining a specific event description language based on CORBA IDL along with mappings for a variety of common programming languages.

Acknowledgments

We would like to thank Martin Odersky for always being at our disposal for questions concerning generic types in general and its application to JAVA in particular, as well as practical issues concerning GJ.

References

[Ada95] International Organization for Standardization. *Ada 95 Reference Manual - The Language - The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[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.

[AFM97] O. Agesen, S.N. Freund, and J.C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, pages 49–65, October 1997.

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

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

[BJ87] K.P. Birman and T.A Joseph. Reliable communication in presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and Ph. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 183–200, October 1998.

[BR97] G. Baumgartner and V.F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):153–187, January 1997.

[BW98] M. Buchi 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 18–22 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, http://www.cs.colorado.edu/ carzanig/papers/, December 1998.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *Conference Record of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90)*, pages 125–135, 1990.

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

[CS98] C. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 201–215, October 1998.

[Dee94] S. Deering. Internet multicasting. In *ARPA HPCC 94 Symposium*. Advanced Research Projects Agency Computing Systems Technology Office, March 1994.

[DEK99] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[EGS00] P. 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)*, June 2000.

[ES92] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1992.

[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)*, 7:80–112, January 1985.

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

[GJS96] J. Gosling, B. Joy, and G. Steele. The Java language specification. Technical report, Sun Microsystems Inc., 1996.

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

[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.

[HLS97] T. Harrison, D. Levine, and D.C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, pages 184–200, October 1997.

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

[KKT96] S. K. Kasera, J. Kurose, and D. Towsley. Scalable reliable multicast using multiple multicast groups. Technical Report UM-CS-1996-073, University of Massachusetts, Amherst, Computer Science, October 1996.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, Ch. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242. June 1997.

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

[LBR96] K. Läufer, G. Baumgartner, and V.F. Russo. Safe structural conformance for java. Technical Report CSD-TR-96-077, Dep. of Computer Sciences, Purdue University and West Lafayette, IN, December 1996.

[Lea97] D. Lea. Design for open systems in Java. In *2nd International Conference on Coordination Models and Languages*, http://gee.cs.oswego.edu/dl/coord/, 1997.

[Mad99] O.L. Madsen. Semantic analysis of virtual classes and nested classes. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 114–131, November 1999.

[MBL97] A.C. Myers, J.A. Bank, and B. Liskov. Parameterized types for Java. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, New York, NY, January 1997.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice-Hall, 1992.

[Mic99] Microsoft. *DCOM Technical Overview (White Paper)*, 1999.

[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)*, April 2000.

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

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

[OMG98a] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, February 1998.

[OMG98b] OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service*. OMG, December 1998.

[OMG98c] OMG. *Notification Service - Joint revised submission*. OMG, January 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.

[OW97] M. Odersky and Ph. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159, Paris, France, 15–17 January 1997.

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

[Rei91] M. Reiser. *The Oberon System*. ACM Press, 1991.

[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*, 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.

[SA98] J.H. Solorzano and S. Alagic. Parametric polymorphism for Java: A reflective solution. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 216–225, October 1998.

[SCC+93] Y.-P. Shan, T. Cargill, B. Cox, W. Cook, M. Loomis, and A. Snyder. Is multiple inheritance essential to OOP? (Panel). In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 363–363, September 1993.

[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.

[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.

[SSLB97] P. Sanjoy, K. Sabnai, J.C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[Sun99a] Sun. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*, 1999.

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

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

[Sun99d] 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*, 10, June 1997.

[Sym97] D. Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, June 1997.

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

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.

[TT99] K.K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In

Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), pages 186–204. June 1999.

[VHK97] J. Vitek, R.N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, pages 142–157, October 1997.

[Vis98] Visigenic. *Naming and Event Services Programmer's Guide 3.2*. Visigenic Software, Inc., Feb 1998.

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