# Abstractions for Distributed Interaction: Guests or Relatives?*

Christian Heide Damm[1], Patrick Thomas Eugster[2], and Rachid Guerraoui[2]

[1] Computer Science Department
University of Aarhus, Denmark
[2] Distributed Programming Laboratory
Swiss Federal Institute of Technology, Lausanne

**Abstract.** *What* abstractions are useful for expressing distributed interaction? This question has constituted an active area of research in the last decades and several candidates have been proposed, including *remote method invocation*, *tuple spaces* and *publish/subscribe*. *How* should these abstractions be supported? Through a library or "directly" within a language? This important complementary question has sparked less enthousiasm.

This paper contributes to addressing this question in the context of Java and the type-based publish/subscribe (TPS) abstraction, an object-oriented variant of the publish/subscribe paradigm. We compare our three implementations of TPS, namely in (1) an extension of Java we designed to inherently support TPS, (2) standard Java, and (3) Java augmented with genericity.

Through our comparison, we identify some general purpose that features that an object-oriented language should have in order to enable a satisfactory library implementation of TPS. We (re-)insist here on the importance of providing both genericity and reflective features in the language, and point out the very fact that the way these features are currently supported might indeed enable satisfactory implementations of remote method invocations, yet is still insufficient for TPS and tuple spaces.

## 1 Introduction

**Motivations.** Typically, when useful programming abstractions are identified, they are provided as libraries written in various programming languages. Sometimes, they are later on integrated into the programming languages themselves. A seminal example is the *monitor* abstraction proposed by Hoare. It was first implemented as a library, became a first class construct in languages like Concurrent Pascal [5] or Portal [9], and is today part of the Java [24] root object type. The same historical pattern has also been applied to abstractions for concurrent, parallel, and most recently, distributed programming. The *remote procedure call*

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

(RPC) abstraction made it through its *remote method invocation* (RMI) incarnation to several languages (e.g., Argus [28], CLU [29], Modula-3 [12], Obliq [11], and Java [40]). A derivate of the *distributed shared memory* (DSM) paradigm appeared as *tuple space* (TS) in Linda [22], and, more recently, we proposed an extension of Java with the *type-based publish/subscribe* (TPS) abstraction [19]. Roughly speaking, TPS is to publish/subscribe what remote method invocation is to RPC: namely, an object-oriented variant of the paradigm.[1]

On the one hand, extending languages with new abstractions is not a priori a good idea, especially in an area like distributed computing where it is still not clear *what* the fundamental abstractions really are. Besides the fact that the language might end up being very complicated, the integration makes any change to the abstraction's implementation usually impossible (the difficulty of this task of course depends on the language). On the other hand, one might argue that integrating an abstraction into a language is legitimated by good arguments, such as type safety or performance.

The goal of this paper is neither to argue for the power of TPS, nor to argue for its integration into Java, but rather to better quantify how much we lose by "simply" supporting TPS as a library implemented on top of the language (with respect to integrating it into the language). Beyond this exercise, we actually address a more fundamental question: what general features should an object-oriented programming language have in order to enable satisfactory library implementations of abstractions for distributed interaction like TPS? Interestingly, and as we discuss in this paper, TPS is demanding enough that identifying those language features means identifying adequate language features that would also enable satisfactory implementations of TS or RMI.

**Contributions.** This paper compares three implementations of TPS. The first implementation is based on Java$_{PS}$ [17], which is a variant of Java we devised with specific primitives for supporting the TPS interaction style. The second implementation [18, 16] is based on standard Java. The third implementation is based on Generic Java (GJ) [4], an extension of Java that provides *genericity* (and is underlying Sun's efforts for integrating genericity into a future version of Java).

We consider four comparison axes: (1) *simplicity*, (2) *flexibility*, (3) *type safety*, and (4) *performance*. Intuitively, these depict (1) the effort invested by a developer when learning how to use the considered TPS solution, (2) the way the considered solution enables developer customizations, (3) the type safety provided when deploying a TPS application, and (4) the performance observed.

Not surprisingly, and even if it somehow impacts flexibility, the Java$_{PS}$ implementation comes off best. However, the GJ implementation seems to offer a close number of positive arguments. Its weak points, compared to Java$_{PS}$, mainly result from its unsatisfactory expression of *content filters* (constraints expressed on events).

[1] Just like RPC, TPS can be integrated with a programming language, yet can as well be implemented in a way which enforces interoperability (à la CORBA).

We use our comparison to point out how inherent reflective and generic capabilities could actually enable a satisfactory library implementation of TPS (and other abstractions for distributed interaction), refraining from any language extensions. While the importance of these capabilities has already been pointed in other contexts, this paper (1) argues, through TPS and Java, that current support of the capabilities in mainstream object-oriented languages is still not sufficient for distributed computing, and (2) attempts to quantify "how much" is currently missing.

**Roadmap.** The rest of the paper is organised as follows. Section 2 briefly recalls the TPS paradigm. Section 3 contains a short introduction to our three implementations of TPS, including some examples of their uses. Sections 4-7 examine the approaches according to our four comparison axes. Section 8 discusses selected design alternatives and the associated trade-offs with respect to our comparison axes. Section 9 overviews language mechanisms supporting powerful library implementations of TPS in a general context. Section 10 discusses related work on distributed programming abstractions, focusing on Java. Section 11 summarizes our paper and draws some conclusions.

## 2 Type-Based Publish/Subscribe

The basic publish/subscribe paradigm offers the illusion of an omnipresent "software bus" interconnecting components in a distributed application, leading to the decoupling of these components.

### 2.1 A Brief History of Publish/Subscribe

Several commercial products support large-scale distributed event-based communication based on the publish/subscribe paradigm (e.g., TIB/Rendezvous [43], SmartSockets [42], iBus [2]). These are all mainly based on the traditional *subject-based* (*topic-based*) publish/subscribe interaction style, in which publishers publish events to a particular subject, and subscribers subscribe to subjects and thus receive the events that are published to those subjects. Most of these systems support one or several specifications out of a proliferating family of standards, e.g., Java Message Service (JMS) [25], CORBA Event & Notification Services [34, 33], or even JavaSpaces [20], which all promote some form of first-class communication channel or subject. The *content-based* publish/subscribe variant, whose origins can be found in academia (e.g., Siena [13], Gryphon [1], Jedi [15]) has made its way into most of these systems and specifications. In content-based publish/subscribe, subscribers can refine their subscriptions further by specifying that they are only interested in events with certain runtime *properties*, these properties usually being interpreted as the attributes of the events.

### 2.2 From Publish/Subscribe to Type-Based Publish/Subscribe

TPS [19] is a recent object-oriented variant of the publish/subscribe interaction style. In TPS, publishers generate and publish instances of native types, i.e., *event objects*, and subscribers subscribe to particular types of objects. A subscription can furthermore have a *content filter* associated, which is based on the public members of the type, including attributes as well as methods. Since event objects are instances of application-defined types, they are first-class citizens. The main contract that the design of such types involves is the subtyping of a basic event type.

**A general abstraction.** TPS is general, in the sense that it can be used to implement the traditional content-based publish/subscribe, and hence also subject-based publish/subscribe. When implemented in a single language, TPS can exploit the type system of the language at hand. TPS can, however, also be put to work in a heterogeneous environment, e.g., by making use of a language-independent event definition language and following a similar approach as the *value types* in CORBA.

**A challenging abstraction.** By enabling the expression of content-based queries based on event methods, TPS offers new possibilities, but also poses new challenges related to the native language connection. Design issues include how to translate the action of "subscribing to a type", and how to express type-safe content filters in the programming language itself, in a way that does not violate encapsulation of events, yet allows for optimisations when applying these filters. Clearly, TPS mainly aims at ensuring [17] (1) *type-safety* and (2) *encapsulation* with (3) *application-defined event types* (the first two requirements could be trivially satisfied with predefined event types). Since TPS aims at large-scale, decentralised applications in which performance is a primary concern, (4) "*open*" *content filters* are important to enable optimisations in the filtering and routing of events: the underlying communication infrastructure must be granted insight into subscription criteria (i.e., content filters) to possibly transfer (parts of) these filters to remote hosts, where they can be applied more efficiently. Last but not least, a form of (5) *qualities of service* (QoS) *expression* is crucial in any distributed context where partial failures are an issue and application requirements on this issue change drastically.

### 2.3 Running Example

We describe below an example application, which is used throughout this paper to examine how our three implementations handle the challenges posed by TPS.

A stock market publishes *stock quotes*, and stock brokers subscribe to these stock quotes. A stock quote is an offer to buy a certain amount of stocks of a company at a certain price, and it may be implemented as shown in Figure 1.

```
public class StockQuote implements Event {
  private String company;
  private float price;
  private int amount;
  public String getCompany() {return company;}
  public float getPrice() {return price;}
  public int getAmount() {return amount;}
  public StockQuote(String company, float price, int amount) {
    this.company = company;
    this.price = price;
    this.amount = amount;
  }
}
```

**Fig. 1.** Simple stock quote events

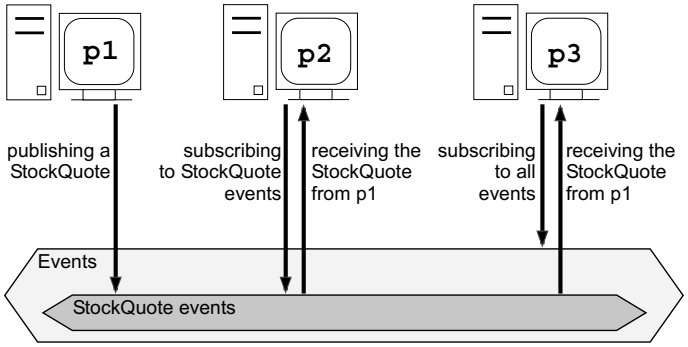


**Fig. 2.** Type-based publish/subscribe

Figure 2 illustrates a situation, where process p1 publishes a stock quote, i.e., an instance of the type `StockQuote`. Process p2 has subscribed to the `StockQuote` type and thus receives the stock quote published by p1. Process p3 has subscribed to the `Event` type, which is the basic event type and a supertype of `StockQuote`, and it thus receives all published events, including the stock quote from p1.

In the examples given in the rest of this paper, we will be interested in stocks from the Telco Group that cost less than 100$. Given a stock quote `q`, this interest can be expressed as follows:

```
q.getPrice()<100 &&
q.getCompany().indexOf("Telco")!=-1
```

I.e., we are interested in the stock quotes, whose company has "Telco" as a substring, and whose price is less than 100.

## 3 Three Implementations

This section gives a short introduction to the three implementations of TPS that we have considered (details can be found in [18, 19, 16, 17]). The first approach augments Java with primitives for TPS, resulting in a dialect of Java called Java$_{PS}$. The second approach is an implementation of TPS in standard Java, while the last approach is based on GJ, which adds *genericity* to Java, as foreseen for Java version 1.5.

### 3.1 Java$_{PS}$ Implementation

Java$_{PS}$, introduced in [17], is a dialect of Java that is designed specifically to support TPS. As such, Java$_{PS}$ represents a proposal for the *optimal* TPS abstraction.

**Syntax.** Java$_{PS}$ integrates TPS by adding two new primitives to the original Java language.

`publish` *Expression*`;`
`subscribe (`*EventType Identifier*`)` *Block Block*`;`

We have made use of an extensible compiler [44] to cope with these primitives. As intuition suggests, the `publish` primitive publishes an event. The `subscribe` primitive generates a subscription to an event type, specifying a first block representing a content filter referring to the actual event through an identifier, and a second block representing an event handler which is executed every time an event passes the filter, using the same identifier. The `subscribe` primitive returns an expression of type `Subscription`, representing a handle for that subscription.

**Programming with Java$_{PS}$.** Using these primitives, a stock quote can be published like the following:

```
StockQuote q = new StockQuote("TelcoOperators", 80, 10);
publish q;
```

Subscribing to stock quotes can be expressed as follows:

```
Subscription s = subscribe (StockQuote q)
  {
    return (q.getPrice()<100 &&
            q.getCompany().indexOf("Telco")!=-1);
  }
  {
    System.out.println("Got offer: " + q.getPrice());
  };
s.activate();
```

As mentioned, the first block specifies the content filter, saying that the subscriber is only interested in cheap Telco stocks. Please note that the content filter is expressed with the exact same code as in Section 2.3 above, which is not obvious, given the required "transparency" of these filters. As we shall see with the other two implementations, expressing content filters in a convenient way is one of the big challenges of TPS.

### 3.2 Java Implementation

Most systems mentioned in Section 2.1 are implemented with language bindings to standard Java. These systems promote a first-class abstraction of a ubiquitous communication channel.

**Distributed Asynchronous Collections.** Similarly, our Java implementation described in this section and in [18, 16] is based on our *Distributed Asynchronous Collections* (DACs). In fact, DACs are abstractions of object containers, which however differ from conventional collections by being asynchronous and essentially distributed. It is thus not centralized on a single host, and operations on a DAC may be invoked through local proxies from various nodes of a network. A DAC appears like a conventional collection (e.g., you can query the DAC with the `contains(Object)` method). A DAC may, however, also be used in an asynchronous way, which means that, instead of invoking the synchronous `contains(Object)` method, you can invoke the `contains(Subscriber,...)` method passing a callback object, which will be notified whenever a new matching element is inserted into the DAC (cf. Figure 3). This kind of asynchrony is similar to the notion of *explicit future* [30], where a method invocation does not return a result immediately, but instead returns a *future object*, which is an object that can be queried for the result later.

Expressing ones interest in receiving notifications whenever an object is inserted into a DAC can be viewed as subscribing to the objects, or *events*, belonging to that DAC. Similarly, inserting objects into a DAC can be viewed as publishing those events, since all subscribers will be notified of the new event. In this sense, a DAC may represent a subject, and publishing and subscribing to events corresponds to inserting events and expressing interest in inserted events, respectively. By mapping types to subjects, a DAC can be used to support TPS. We use a scheme for translating types to subjects, which encodes the type and class hierarchy of the type in a subject based on a URL-like notation. For example, the event class `StockQuote`, which extends `Object` and implements `Event` (which in turn extends `java.io.Serializable`), will be represented by the following string (for simplicity we have omitted the package names of the `StockQuote` and `Event` types):

```
java.lang.Object/StockQuote(Event(java.lang.Serializable))
```

```
interface DAC extends java.util.Collection {
  boolean add(Object event);
  Object get();
  boolean contains(Object event);
  boolean contains(Subscriber subscriber, Condition contentFilter);
  ...
}

interface Subscriber {
  void notify(Object event, String subject);
}

interface Condition {
  boolean conforms(Object event, String subject);
}
```

**Fig. 3.** Basic interfaces in the Java implementation

**Subscribing with DACs.** A subscription to an event type (and implicitly, its subtypes) is issued through a DAC representing that type, which might require the creation of a new DAC for that type if none is available.

Returning to the example application described in Section 2, Figure 4 illustrates how a stock broker issues a subscription. The subscriber subscribes to the DAC representing the type `StockQuote`, and assumes both in the `notify()` method and in the content filter that events are of that type (the instantiated DAC class `DAS` reflects reliable delivery). The awkward appearance of the filter is motivated by the special requirements on content filters, such as its undergoing of deferred evaluation to enforce prior optimisation (see Section 7).

**Publishing with DACs.** Similarly, the stock market publishes stock quotes through the DAC representing the type `StockQuote` like this:

```
DAC stockQuotes = new DAS("StockQuote");
StockQuote q = new StockQuote("TelcoOperators", 80, 10);
stockQuotes.add(q);
```

### 3.3 GJ Implementation

In our Java implementation of Section 3.2 just described, a DAC is used to represent a specific type, yet nothing would prevent, at least at the time of compilation, an attempt of inserting non-conformant events into a DAC. Even if all published events inserted into a given DAC are of the right type, the programmer has to manually cast events to the desired type upon receiving them. Using genericity, illegal inserts and manual type casts can be avoided. Many languages support some sort of genericity directly, including C++ [39], Ada 95 [27], and BETA [8], whereas other languages were initially designed without support for genericity in mind, including Java and Oberon [35].

```
class StockQuoteSubscriber implements Subscriber {
  public void notify(Object event, String subject) {
    StockQuote q = (StockQuote)event;
    System.out.println("Got offer: " + q.getPrice());
  }
}

Condition telcoCondition = new Equals("getCompany.indexOf",
  new Object[]{"Telco"}, new Integer(-1));
Condition priceCondition = new Compare(".getPrice",
  new Object[]{100}, -1);
Condition contentFilter = telcoCondition.not().and(priceCondition);

Subscriber subscriber = new StockQuoteSubscriber();
DAC stockQuotes = new DAS("StockQuote");
stockQuotes.contains(subscriber, contentFilter);
```

**Fig. 4.** Subscribing with DACs

**Generic DACs.** The generic library approach described in this section, and introduced in [19], is based on GJ [4], which is an extension of Java with support for genericity through *parametric polymorphism* (*F-bounded polymorphism* [10]). With parametric polymorphism, we obtain *typed* DACs without generating type-specific code, and nevertheless avoid explicit type casts. The resulting generic DACs (GDACs) and associated types are shown in Figure 5. As a result of the typed subscriber, there is no longer a need for a subject name parameter in the callback method of the `GSubscriber`, because the classification is given implicitly by the type.

**Programming with GDACs.** Using this generic version of DACs, stock quotes can be published like this:

```
GDAC<StockQuote> stockQuotes = new GDAS<StockQuote>(StockQuote.class);
StockQuote q = new StockQuote("TelcoOperators", 80, 10);
stockQuotes.add(q);
```

Subscriptions expressed through GDACs come very close to subscriptions expressed with DACs, and we will leave it to the reader to see how the example in Figure 4 can be modified to use GDACs. Please note that the parameter passed to the GDAC constructor above is necessary, since GJ does not provide runtime type information.

As in our Java implementation (Section 3.2), the programmer is responsible for publishing and subscribing to the right GDAC, i.e., the GDAC representing the event type. However, in this case, the compiler assists the programmer in developing type-safe code, by performing type checks (and casts).

```
interface GDAC<T> {
  boolean add(T event);
  T get();
  boolean contains(T event);
  boolean contains(GSubscriber<T> subscriber,
                   GCondition<T> contentFilter);
  ...
}

interface GSubscriber<T> {
  void notify(T event);
}

interface GCondition<T> {
  boolean conforms(T event);
}
```

**Fig. 5.** Basic interfaces in the GJ implementation

The following four sections compare our three implementations, according to simplicity, flexibility, type safety, and performance.

## 4 Simplicity

*Simplicity* is a (subjective) measure of the effort necessary (1) for a programmer to *learn* and *use* the considered implementation of TPS, and (2) for third parties to *read* and *understand* TPS-related code. Clearly, distributed applications can become very complex, and a powerful yet simple programming abstraction can reduce the burden on the developer.

Note that simplicity does not necessarily favour a language integration. Indeed, a programmer acquainted with other publish/subscribe systems might find it easier to shift from one Java library to another, than to learn a "new" language.

### 4.1 Content Filters

An important aspect of simplicity relates to the query language with which subscriptions are expressed – the *subscription language*.

**Subscription language.** In most common publish/subscribe systems providing a form of content-based subscribing, a subscription can contain (besides possibly a URL-like expression representing an explicit subject) simple predicates ($=$, $\neq$, $<$, $>$, etc.) on the attributes of events [13]. These subscriptions are most commonly expressed by a string following a specific grammar (as shown by specifications such as the CORBA Notification Service [33] or JMS [25]).

Such query languages do not necessarily have a negative effect on simplicity. They are even likely to have a very concise syntax and semantics (consider, e.g., SQL for relational databases). However, these languages systematically express queries through attributes, violating encapsulation, and they offer only poor support for queries based on methods, such as required by TPS.[2]

**Programming language.** In our Java$_{PS}$ implementation,[3] the content filters are truly expressed in the native language, making them simple to express for programmers that are already familiar with the native language. There are, however, restrictions on what variables can be accessed inside content filters. Indeed, to make filters easily transferable in a distributed environment, only `final` variables declared outside the filter can be used, and these can only be of primitive object types, such as `Integer` or `Float`, including `String` [17].

**Subscription API.** Our Java and GJ implementations on the other hand introduce a form of subscription language, based partly on an API, and partly on the native invocation semantics of Java. Primitive conditions are reified as `Condition` objects, and are logically combined through method calls on them.

Although subscriptions our the Java and GJ implementations rely on method invocations, these are difficult to express, i.e., even simple constraints lead to poorly readable code (see the `telcoConditions` used in Figure 4). In addition, many errors, e.g., a wrong number of parameters, are only detected at runtime. Clearly, content filters in this subscription scheme enforce encapsulation at a high price in terms of simplicity.

### 4.2 Channel Management

In our Java and GJ implementations, the actions of publishing and subscribing are both expressed through first-class channel abstractions, (G)DACs, which have to be managed explicitly. Though programmers experienced with publish/subscribe systems are used to publishing explicitly through some form of channel or connection abstraction, it is easier to understand for a programmer who is less experienced with publish/subscribe, that objects to be published are simply "fired", and similarly, that subscriptions are expressed arbitrarily in the application.

### 4.3 Qualities of Service

The limited form of QoS expressed through the specific (G)DAC type, e.g., (G)DASet for reliable communication, (G)DAList for additional ordering guarantees (see [18]), enables the use of the same event types with different and

[2] Furthermore, query languages provide little safety, and are plentiful, contradicting the often claimed portability.

[3] This abbreviates "our Java$_{PS}$ implementation of TPS".

maybe even incompatible QoS: a publisher can publish events of a given type through a DAC offering best-effort guarantees, while a party subscribed to that type has expressed its desire for receiving all published instances by subscribing to a DAC reflecting reliable delivery. What would a compromise between these QoS be? With an increasing number of QoS parameters, it becomes difficult to define a general policy for mediating between different expectations. With the current implementations, developers are expected to ensure manually that DACs used with the same type of events are of the same type as well.

This risk of potential mismatch has been strongly reduced in our Java$_{PS}$ implementation by expressing the QoS through the events themselves. In other terms, the QoS associated with an event are given as part of its type, which is in fact anyway the only "contract" between publishers and subscribers. E.g., by subtyping an event type `Reliable`, the developer expresses that all instances of that event type should be reliably transferred between publishers and subscribers.

### 4.4 Receiving Events

In our Java and GJ implementations, a subscriber must implement a `notify()` method, which is invoked upon reception of an event. This method is implemented by a callback object – an event handler – and passed to the (G)DAC upon subscription. The code for such an event handler, i.e., a class that implements (`G`)`Subscriber`, is isolated in a specific class, leading to a scattering of the code related to single subscriptions.

In our Java$_{PS}$ implementation, the above event handler is viewed as a *closure*, whose signature is implicitly given as part of the syntax of the subscription expression, and all the code related to a subscription is colocated, making it easy to understand what the subscription does. Given that the content filter and the event handler are two sides of the same story, it seems more adequate to concentrate these at the same place.

### 4.5 Verdict

Our TPS-specific language primitives in Java$_{PS}$ offer a very concise syntax, and despite the restrictions put on filters, these are easiest to use: subscription expressions are compact and use a subset of native Java syntax, which makes them easily understandable.

The Java implementation, at the other end of the range, presents many potential conflicts, like the possible mismatch in QoS also found in the GJ implementation, but also possible mismatches in types when casting or setting up DACs. Although event handlers in the Java implementation, and also in the GJ implementation, could be expressed as anonymous classes placed inside subscription expressions, the heavy syntax of these anonymous classes (compared to the specific closures in the language integration approach) limits their benefit much in terms of simplicity. In addition, filter expression in these two approaches

suffers from the lack of custom *operator overloading*, which could enable simpler combinations of conditions (see Section 8.2).

# 5 Flexibility

By the *flexibility* of an implementation of publish/subscribe, we mean the extent to which it can be used to devise applications based on (type-based) publish/-subscribe with various requirements.

## 5.1 Content Filters

Content filters make event handling simpler, and increase performance by avoiding sending events to subscribers that anyway ignore them.

All three implementations allow for arbitrarily complex content filters. However, the Java and GJ implementations have a rather cumbersome way of expressing content filters, and it is thus likely that programmers are tempted to shift at least parts of the content filters to the event handlers, with serious consequences on performance. This is slightly counterbalanced by giving developers the possibility of writing their own conditions; only slightly, because in order to nevertheless enforce optimizitions, such custom conditions must provide several hooks to enforce optimizations (see Section 8.2).

In our Java$_{PS}$ implementation, it makes no difference to the programmer if the filtering is done in the content filter or in the event handler, since these are expressed in the same language. By the absence of reified conditions, such as in the Java and GJ approaches, specific conditions can be implemented by integrating their logic into the events, however only prior to deployment.

## 5.2 Qualities of Service

In our Java$_{PS}$ implementation, the quality of service is specified in the type of the event. Although this solution would also have been possible in the other implementations, these associate QoS with the channel abstractions, as it is done in many other publish/subscribe systems. The already mentioned possible conflicts between QoS of publishers and subscribers in this case can diminish simplicity, but potentially increases flexibility. Indeed, much research effort has been invested into negociating between different QoS requirements, and (G)DACs could be implemented in a way which enables participants to use diverging (G)DAC types to connect to the same event types. In Java$_{PS}$, a weakening of QoS requirements could be expressed, less elegantly, by adding methods to the subscription handles returned by the `subscribe` expression.

The QoS framework used in the Java and GJ implementations can itself be more easily extended, by adding, deriving, and combining new (G)DAC types, since these reflect the guarantees they offer. In our Java$_{PS}$ implementation, such a customisation becomes more difficult. Although new abstract event types similar to `Reliable` etc. can be added to the framework to reflect new kinds of services, these types are decoupled from the actual algorithms implementing them.

Any extension of the QoS framework hence currently requires the intervention of one of its developers, while simple customizations in the case of (G)DACs can be more easily made by any experienced developer.

### 5.3 Verdict

Obviously, identifying *all* possible application scenarios for TPS is impossible. Should there arise new needs at some point, which require changing the publish/subscribe system, a library in Java or GJ is easier to change. For instance, none of the three implementations currently addresses security aspects. Restricting the scope of an event type would require modifications to (G)DACs, but also to Java$_{PS}$. The former case is however easier to handle.

A library will always is typically more flexible than a solution integrated in the language, since the latter type of solution is more tedious to modify.

## 6 Type Safety

Most recent object-oriented programming languages are statically typed, aiding the developer in devising reliable applications. Distributed applications bring an increased degree of complexity, and it becomes even more important here to assist developers by providing them with mechanisms to ensure type safety in remote interactions.

### 6.1 Publishing and Receiving Events

In our Java implementation, publishing an event corresponds to inserting the event into an untyped collection (DAC). It is impossible to ensure at compilation that an event is published through a DAC that represents the type of that event (or a subtype), and symmetrically, there is a high risk that a subscriber casts events to a wrong type. These type coercions strongly contradict our requirements for type safety, since an event consumer might not be able to foresee the types of events that it will recieve. To avoid that the program halts prematurely, runtime exceptions (`ClassCastException` in Java) will have to be caught.

In our Java$_{PS}$ implementation, publishing and receiving events is completely type-safe. In the GJ implementation, both publishing and receiving events is type-safe, provided that the involved GDACs have been correctly initialized: due to the absence of runtime information on type parameters in GJ, a class meta-object is expected by GDAC constructors (see Section 3), which can lead to possible mismatches. In fact, the *homogenous* implementation of genericity in GJ "erases" type parameters, making it impossible for an instance of a type parameterized class to find its actual type parameters. The need for such *runtime support for type parameters*, has also become apparent in other contexts, e.g., orthogonal persistence [37].

### 6.2 Content Filters

The content filters in our Java$_{PS}$ implementation are completely type-safe, since they are type-checked by the compiler. In the other two implementations, content filters are expressed partially through strings, putting type-safety at stake. Type checks can however be performed at runtime in predefined content filters (e.g., `Equals` and `Compare`, see Section 3.2), through the *introspection* capabilities of Java. For instance, it can be ensured that the `StockQuote` class implements methods corresponding to the name and arguments that are passed to the constructor of a condition (e.g., `".getCompany.indexOf"`).

Note, however, that the developer, though not using reflection explicitly to define *which* methods (and arguments) are to be used to query events, has to be aware of the fact that reflection is used underneath to find the appropriate methods: unlike with static invocations in Java, the dynamic types of the specified invocation arguments are used to identify the appropriate method.

### 6.3 Verdict

To summarise, and not surprisingly, the safety increases in the GJ implementation compared to the Java implementation, and it increases further with Java$_{PS}$, where there can be no "type unsafety" related to publish/subscribe.

Although the GJ implementation ensures type safety when publishing and receiving events, this is probably of less importance than having type-safe content filters. The reason is that most Java programmers are currently used to having untyped collections: they know that they must insert `StockQuote`s in the collection referred to by `stockQuotes`, and they know that they have to cast the result when extracting an element from such a collection. Having typed content filters, on the other hand, is really useful, because here you may make a lot of errors.

## 7 Performance

Last but not least, we present the most significant results of our performance measurements realized with the three different approaches [31]. We actually measure the overhead of the GJ and Java approaches with respect to Java$_{PS}$.

### 7.1 Setting

In comparing the different approaches, we have used the same simple architecture underlying our first library implementation in Java as a testbed. That architecture is characterized by a *class-based* dissemination, i.e., every event class is mapped to an IP Multicast channel. The test application involved three types; a type `Event`, its subtype `StockQuote`, and a subtype of the latter type, `StockRequest`. Since the filter evaluation seen from a system perspective is essentially the same in all three approaches, we have focused more on the static aspect of TPS.

The measurements presented here concentrate on the *latency* of publishing events, which refers to the average time (ms) that is required to publish an event onto the corresponding channel (as perceived by the publisher), including a 1ms break between each two events. Further measurements are presented in [31], e.g., illustrating the impact of latency and other aspects on *reliability*, i.e., a measure of the percentage of events that are delivered correctly to their intended subscribers.

### 7.2 Library vs Language Integration

The two library implementations differ from the implementation of Java$_{PS}$, in that upon publishing an event, the precise channel for the corresponding class has to be found. In the case of Java$_{PS}$, a simple `publish()` method is automatically added to every event class, which automatically pushes the event onto the fitting channel.

This difference is visible in Figure 6, where we compare the GJ implementation (the Java implementation yielded similar results) with our Java$_{PS}$ implementation. One can see that the latency of publishing an event in the case of GJ is increased by runtime type checks performed to obtain the appropriate channel. The latency varies here with the number of events published in a row (due to a "warm-up" effect observed with IP Multicast). As the figure conveys, the difference in latency remains nearly the same with a varying number of published events.

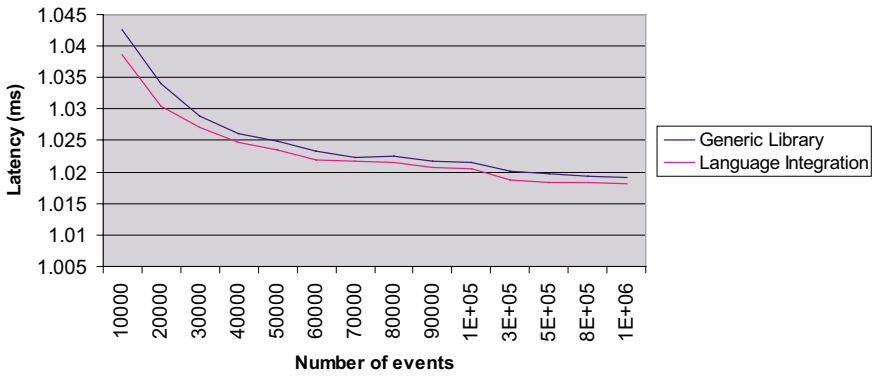


**Fig. 6.** Latency of publishing: Java$_{PS}$ vs GJ

### 7.3 The Cost of Subtyping

The performance of the library approaches is conditioned by the number of different subtypes whose instances are published through a given (G)DAC. The

second set of latency measurements presented here relates to the GJ implementation, and intends to compare the latencies obtained with the various event types published through a GDAC for the uppermost type. Figure 7 conveys the very fact that the system performs best for the uppermost type of the hierarchy (`Event`) and that the performance degrades as we go down this hierarchy. This was expected, since publishing a `StockEvent` through a GDAC for type `Event` in our architecture involves a lookup of the corresponding channel in an internal structure (and possibly the creation of the channel). This lookup in the case of the `StockRequest` type, requires even more effort.

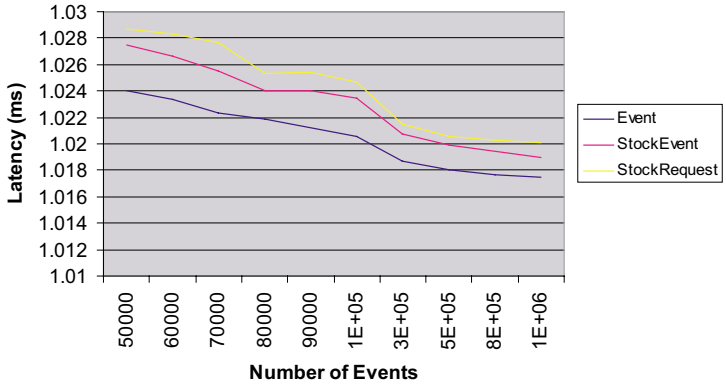


**Fig. 7.** Latency of publishing different event types

### 7.4 Verdict

The latency observed when publishing events is slightly, but clearly, smaller in the case of Java$_{PS}$ than in the case of the Java or GJ implementations. Also, this increased latency becomes even more important as the events published through a (G)DAC are of an increasing number of different subtypes of the event type represented by that (G)DAC. Note, however, that optimisations for the involved channel lookups could certainly be performed (more details are given in [31]).

## 8 Trade-Offs

Some of the differences between the features of the three implementations reflect explicit design choices, while others result from inherent limitations. This section first summarises the results from the previous sections, and then discusses a selected choice of trade-offs related to explicit design choices.

### 8.1 Comparison Summary

Table 1 summarises the results of the previous sections. Clearly, our Java$_{PS}$ implementation comes off best, with the GJ implementation coming in second. The weak points of the GJ implementation mainly result from its unsatisfactory expression of content filters.

| | Java | GJ | Java$_{PS}$ |
|---|---|---|---|
| Simplicity | $\sim$ | $\sim$ | $+$ |
| Flexibility | $+$ | $+$ | $\div$ |
| Type safety | $\div$ | $\sim$ | $+$ |
| Performance | $\sim$ | $\sim$ | $+$ |

**Table 1.** Comparison summary ($\div$ insufficient, $\sim$ acceptable, $+$ good)

### 8.2 Content Filters

Many controversies reside around the largely unsatisfactory content filter expression in the Java and GJ implementations.

**Simplicity vs performance.** One way to improve simplicity of content filter expression, as already alluded to, consists in putting user-defined (public) condition classes to work, instead of predefined ones. The condition in the stock quote example could then be implemented as illustrated by Figure 8. The `conforms()` method defined in class `PriceAndCompanyCondition` contains a very concise and readable form of the same filter expressed previously in Section 3.2.

Regardless of the fact that predefined conditions are reusable and probably sufficient for expressing many simple filters, user-defined conditions present the considerable shortcoming that they are difficult to optimise. In the absence of the source code of used-defined filters, for instance, automatic optimisations such as the avoiding of redundant invocations on the events (cf. [16]), require the condition implementer to be accustomed with reflection in Java, in order to implement the required hooks.[4]

The same problem occurs when trying to merge the event handler and filter in the same class: it is hard to impose restrictions on the filter semantics and to supply the underlying communication engine with an insight into such filters.

[4] Optimisations could to some degree be performed on byte code, but there it becomes difficult to impose restrictions on what can be done inside a content filter, which is very important to limit the amount of code that has to be sent over the wire when transferring such a filter to apply it at a more favourable stage.

```
public class PriceAndCompanyCondition implements Condition {
  private float priceLimit;
  private String companySubstring;
  PriceAndCompanyCondition(float priceLimit, String companySubstring) {
    this.priceLimit = priceLimit;
    this.companySubstring = companySubstring;
  }

  public boolean conforms(Event e) {
    StockQuote q = (StockQuote)e;
    return (q.getPrice()<priceLimit &&
            q.getCompany().indexOf(companySubstring)!=-1);
  }
}
...
DAC stockQuotes = new DAS("StockQuotes");
stockQuotes.contains(..., new PriceAndCompanyCondition(100,"Telco"));
```

**Fig. 8.** User-defined conditions in the Java implementation

**Type safety *and* simplicity.** An alternative mechanism for the type-safe expression of content filters was anticipated with the integration of *behavioural reflection* with Java 1.3: while the introspection mechanisms present in Java since version 1.1 enable the reification of methods and the dynamic invocation of these, behavioural reflection allows the interception of also statically expressed method invocations, and the performing of pre- and post-invocation actions. Such a scheme could allow the developer to express queries on event objects by making the corresponding invocations on a proxy, which *records* these invocations (reifies the invocations), such that they can be *replayed* on the effective events. To that end, the `contains()` method of the (`G`)`DAC` interface would be changed, as illustrated by Figure 9, to return an instance of a proxy class bound to an `InvocationHandler`, which would register the invocations performed on it.

However, Java's `Proxy` class only allows this as long as the effectively imitated (event) type `T` – and in the case of nested invocations also the types of any object returned as result of an invocation – are interface types. This precludes the use of primitive types, and also primitive object types (e.g., `Integer`). Hence, the GDAC implementation sketched in Figure 9 could not be instantiated with strings, making even the simple following scenario impossible:

```
class StockQuoteSubscriber implements GSubscriber<StockQuote> {...}
GDAC<StockQuote> stockQuotes = new GDAS<StockQuote>(StockQuote.class);
StockQuote quote = stockQuotes.contains(new StockQuoteSubscriber());
quote.getCompany().equals("Telco");
```

Furthermore, the composition of composed filters, such as

```
String company = quote.getCompany();
company.equals("Telco") || company.equals("MobileCo");
```

would require operators defined on primitive types to be reflected by methods on the corresponding object types, which sometimes comes as part of operator overloading not currently part of Java.

```
public class GDASet<T> {
  private Class type;
  ...
  public GDASet(Class type) { this.type = type; }
  ...
  /* return a "fake" instance of T
  public T contains(Subscriber<T>) {
    return (T)Proxy.newProxyInstance(type.getClassLoader(),
                new Class[] { type }, new FilterRegistrar());
  }
  ...
  private static class FilterRegistrar implements InvocationHandler {
   public Object invoke(Object proxy, Method method, Object[] args) {
        /* register the invocation */
        ...
        /* if the method returns an object, return another proxy */
        return ...;
    }
  }
}
```

**Fig. 9.** Using behavioural reflection for content filter expression

### 8.3 Improving Simplicity in the Java and GJ Implementations

In the Java and GJ implementations, the programmer has to create a (G)DAC for the right type in order to subscribe to that type. As discussed in Section 4, this reduces simplicity. As the (G)DAC depends directly on the event type, it seems unnecessary to require the programmer to create the (G)DAC. Figure 10 illustrates how this could be circumvented by a wrapper, which would be used like this:

```
Condition condition = new Compare(".getPrice", ....);
Subscriber subscriber = new StockQuoteSubscriber();
pubsub.subscribe(subscriber, StockQuote.class, condition);

pubsub.publish(new StockQuote("TelcoOperators", 80, 10));

pubsub.unsubscribe(subscriber, StockQuote.class);
```

Of course, the wrapper could be implemented to be more efficient than this by reusing the created DAC proxies between calls, and QoS could be expressed, akin to Java$_{PS}$, through the event types.

This also removes the possibility of publishing an event to a wrong DAC, i.e., type safety is improved. Such wrapper methods can also be implemented in GJ, since methods can also be implemented in a generic manner. However, reusing the same GDACs poses problems, due to the diverging type parameters (when storing GDACs with different type parameters in a collection, the type parameters are lost).

```
class pubsub {
  static void subscribe(Subscriber subscriber, Class type,
                        Condition contentFilter) {
    DAC dac = new DAS(type.getName());
    dac.contains(subscriber, condition);
  }

  static void unsubscribe(Subscriber subscriber, Class type) {
    DAC dac = new DAS(type.getName());
    dac.clear(subscriber);
  }

  static void publish(Event e) {
    DAC dac = new DAS(e.getClass().getName());
    dac.add(e);
  }
}
```

**Fig. 10.** Wrapping DACs

## 9 Language Mechanisms for TPS

What language features would enable a library implementation of TPS (and maybe also other paradigms for distributed programming) in a way that satisfies our requirements outlined in Section 2.2, without any extensions to the target language? We address here this question using elements of our comparison in the previous sections.

### 9.1 Type Safety

Type safety has to be ensured mainly at two points. First, upon subscriptions, it has to be guaranteed that filters and handlers deal with the same type of events, which is different for every subscription. Hence, a subscription is type parameterized, requiring some form of *genericity*. Second, given the nature of our distributed context, there must be a way for distributed participants making use of the same event types to "connect". This requires runtime type information,

e.g., the possibility of reifying types. Indeed, verifying how types are related, and performing runtime type inclusion checks on objects ensures type safety at the communication infrastructure level. Such mechanisms are commonly viewed as part of *introspection*, or more generally, *structural reflection*.

As shown by the GJ implementation, *runtime support for type parameters* in genericity can be useful to perform dynamic type checks if the filters can not be checked at compilation.

### 9.2 Filters

In Java$_{PS}$, filters are implemented by some form of *deferred code evaluation* (e.g., MetaML [41]) to ensure that these can be type-checked at compilation, by offering an insight to the middleware [17]. A reification of the entire program in the form of *parse tree*, such as in Smalltalk [36], could address the same requirement. Alternatively, *behavioural reflection*, possibly combined with *operator overloading*, could provide for an ideal compromise of static type checking and deferred evaluation.

The requirements posed by filters are the most tedious to fulfil, and only few languages provide sufficient mechanisms.

### 9.3 Handlers

Handlers can be methods implemented by callback objects of a specific type, or *closures* (also *function pointers*). While closures enable the concentration of all subscription-related code, pointers enable the placing of the event handler at any point, and methods force the definition of a specific class. With respect to type safety, the main issue consists in verifying that the signature of the provided piece of code corresponds to the filter, i.e., the handler and the filter each have a single formal argument with a coinciding type. In the case of a callback object, to avoid generating specific callback types for each event type, *genericity* can be used to type parameterize the callback type.

### 9.4 Events

*Serialization*, provided as inherent feature in Java, is a first step towards supporting a programming scope exceeding a single address space. Events, in all the implementations of TPS, exploit this mechanism in Java. In languages lacking such a built-in mechanism, the application developer can be asked to implement hooks to support serialization/deserialization of the events. The Java model (inspired by Smalltalk), however, is very simple to understand and to use, since it provides a default behavior which can be overridden if needed.

Subtyping, a key paradigm in object-oriented programming, provides the necessary foundation for the ability of updating applications. Especially in a distributed context, the full exploitation of this paradigm is only given by *dynamic class loading*, that is, the possibility of updating existing components by adding new implementations at runtime.

### 9.5 No Limits

The prominent mechanisms stated above have all been investigated in the context of TPS in variants of Java, and by no means the intention is to claim that the outlined mechanisms cover all possibilities, since there are obviously many other language features to think about.

Consider for instance the `myType` type qualifier introduced by Bruce et al. in PolyTOIL [7], and inherited by its follow-up Loom [6]. In any given method body, this qualifier refers to the dynamic type of the considered object, the type of `this`. In the words of the authors, "`myType` is anchored to the type of the object in which it appears". This paradigm enables an inherently clean implementation of binary methods. In the context of TPS, it could be used in combination with behavioural reflection and simple unbounded parametric polymorphism (also part of PolyTOIL), to ensure type-safe *direct subscriptions* to application-defined event classes (i.e., without first-class channels), which subtype a specific root event type `Event`. Following the Java syntax, one could imagine having something like the following:[5]

```
public class Event {
  public myType subscribe(Subscriber<myType> s) {
    ...
    /* return a proxy */
  }
}
public class MyEvent extends Event {...}
```

Subscribing to an application-defined event class, such as the `MyEvent` class above, can be done simply by first creating an instance of that class, and then invoking the `subscribe()` method:

```
class MyEventSubscriber implements Subscriber<MyEvent> {...}
MyEvent proxy = new MyEvent().subscribe(new MyEventSubscriber());
proxy.equals(new MyEvent(...));
```

The above subscription scheme could fulfil all our requirements in a language which, unlike Java, does not provide any purely abstract types (since these are not supported by the above design). Furthermore, if the `myType` type qualifier is available in class methods, one could omit creating an instance of an event class just for subscribing to that type.

Similarly to the `myType` type qualifier, the concept of *mixins* could enable the merging of the abstraction for subscribing with the very event types; here by "adding" methods expressing subscriptions and unsubscriptions to application-defined event types instead of inheriting them from an abstract event type.

While the investigation of further candidate language features is an ongoing task, the experience related throughout this paper entitles us to claim that Java clearly does not fulfil our requirements with respect to TPS.

[5] `myType` has a companion, written `@myType`, which denotes the *exact* dynamic type of `this`, i.e., subtypes are considered harmful. To be fully precise, the following example should use `@myType` as type parameter for the subscriber.

## 10 Related Work

We are only aware of one effort discussing different ways of integrating publish/subscribe into a language, namely the *events + constraints + objects* (ECO) model [26]. In that context however, the question of what language mechanisms would help avoiding any extension is devoted less attention.

In the following, we look at the way this question was addressed for two alternative distributed interaction abstractions with respect to Java, namely the tuple space (TS) and the remote method invocation (RMI) paradigms.

### 10.1 Tuple Spaces

The TS abstraction first appeared in the Linda programming language [22], in which spaces served as coordination means between cooperating processes.

**The basic abstraction.** A TS is a place where processes can exchange arbitrary length tuples of values. Putting a tuple into the TS is done using the `out` primitive. Getting a tuple from the tuple space is done using a blocking primitive, either `in` to subsequently remove the read tuple from the space, or `read` to enable the same tuple to be read by several consumers. The TS has since been extended with further primitives, e.g., non-blocking read primitives and callbacks. The latter option leads to a publish/subscribe-like interaction when combined with non-destructive (`read`) semantics. Consider the following example expressed in Linda:

```
out ("StockQuote", "Telco", 80, 10);           // 1
int i = 80;                                    // 2
in  ("StockQuote", "Telco", i, 10);            // 3
in  ("StockQuote", "Telco", var i, 10);        // 4
in  ("StockQuote", "Telco", j: integer, 10);   // 5
```

In line 1, a tuple consisting of 4 values is put into the tuple space. In line 3, a tuple with 4 values is requested. Since the value of `i` is 80, the tuple from line 1 matches the request, which means that this tuple may be extracted from the TS. The `var` keyword in line 4 causes the `i` to be treated as a formal parameter, i.e., it can match any value. The tuple added in line 1 may be extracted by line 4, and the actual value of `i` will then be 80. In line 5, the `integer` keyword at the same time declares a variable `j` and uses it as a formal parameter as in line 6.

**Translating to Java.** Implementing TSs in Java poses similar problems to those we described for TPS in this paper. As an example, Jada [14] instruments Java with a library that supports TSs. In Jada, a `Tuple` represents a list of Java `Objects`, i.e., the values of the tuple. Clients thus have to cast these objects explicitly upon reception, thereby reducing type safety. In order to improve simplicity, a `Tuple` has constructors for up to 10 values. Formal parameters are represented by objects representing the desired type (instances of `java.lang.Class`; meta-objects). The above Linda example can be expressed in Jada as follows (note that line 5 has no equivalent in Jada):

```
TupleSpace tupleSpace = new TupleSpace();
tupleSpace.out(new Tuple(
  "StockQuote", "Telco", new Integer(80), new Integer(10))); // 1
int i = 80;                                                  // 2
Tuple tuple1 = tupleSpace.in(new Tuple(
  "StockQuote", "Telco", new Integer(i), new Integer(10)));  // 3
Tuple tuple2 = tupleSpace.in(new Tuple(
  "StockQuote", "Telco", Integer.class, new Integer(10)));   // 4
i = ((Integer)tuple2.getItem(3)).intValue();
```

As illustrated by Jada, such a TS library becomes clumsy compared to the original support in the Linda language.

More recent approaches to TS interaction into Java, like JavaSpaces [20], apply a different model, viewing tuples as single objects, whose attributes reflect tuple values. Hence, the JavaSpace type requires a single signature for its `read()` operation. Custom events are defined by subtyping the basic `Event` type, which again does not ensure type safety, since type checks and type casts are necessary. Also, encapsulation is broken by forcing attributes to be declared as public; expressing and performing any content-based filtering through these attributes.

**Library or language integration?** The question of library vs language integration has also been raised in the context of TSs. Rather surprisingly, a separation of the programming language from the concurrency mechanism was advocated by Carriero and Gelernter [23], while their Linda language is widely viewed as a monolithic solution to merging a coordination language with a programming language.

In the case of Java, through the similarity between TPS and TSs such as JavaSpaces, a "clean" library could be implemented with similar features claimed for TPS.

### 10.2 RMI

Another prominent mechanism for distributed interaction in whose context the question of language integration vs library has been addressed is the RMI paradigm.

**Java RMI.** The implementation of Java RMI can be viewed as an intermediate solution between a language-integrated RPC package and a standard Java library. Java RMI relies on the inherent Java type system, yet further constrains the use of that type system in its own context: (1) static types of remote references must be abstract types, i.e., interfaces, and (2) any methods in such interfaces must imperatively declare that they can throw `RemoteException`s. Java RMI can also be considered as a language extension in the sense that a specific compiler (`rmic`) is needed to generate type-specific proxies. The absence of the corresponding proxies is, however, only signalled at runtime.

**Library or language integration?** A form of RMI can be implemented in Java as a pure library (without specific compilation) with Java 1.3, thanks to its support for behavioural reflection: the same mechanism used in Section 8.2 can be used to defer the binding to a remote object to runtime. This mechanism has obviously been devised with the requirements of RMI in mind. Indeed, (1) the class responsible for behavioural reflection has been called `proxy`, and (2) only interface types can benefit from this type of reflection, and the static types of remote Java objects are always interfaces.

In the case of TPS, where "nested" invocations, i.e., invocations on the return types of invocations, have to be intercepted, the `proxy` class is clearly insufficient, as illustrated in Section 8.2. In the case of RMI, it is not clear whether this mechanism for behavioural reflection will replace the generation of type-specific proxies through the `rmic` compiler. While in the context of interoperability, like in CORBA, such a precompilation can help dealing with language diversity, a precompilation can in the context of Java only more be motivated by performance reasons.

## 11 Conclusions

Java$_{PS}$, our extension of the Java language, was motivated by the obvious lacks manifested by the Java language with respect to TPS. First, to achieve a level of type safety worth mentioning as such, we made use of GJ, a variant of Java incorporating genericity. Second, to enable a satisfactory expression of content filters, we went a step further and devised Java$_{PS}$. Comparing the three approaches helps measure the difference between Java$_{PS}$, on the one hand, and the two library approaches (i.e., Java, and Java with genericity), on the other hand.

In general, and in the face of today's heterogeneity across platforms, we believe that programming languages should not be implemented with abstractions for distributed interaction as primitives. We rather believe that designers of future languages should foresee a more general support for distributed interaction abstractions. In particular, avoiding an integration avoids the question of *which* abstractions for distributed interaction should be supported. Although TPS is surely not the last paradigm for distributed programming, the constraints imposed by TPS should be kept in mind when conceiving future support for distributed programming. As shown by the difficulty in expressing content filters, TPS, as a paradigm emphasizing scalability and performance, requires a strong interaction with the native programming language, and is hence a very demanding abstraction. Most abstractions established for distributed interaction, such as sockets, tuple spaces, or RMI, can be implemented with only a subset of the features mandated by TPS.

We argue that reflection, just like genericity, as faces of *extensibility*, to be the key concepts for a general language support of distributed programming. With inherent reflective capabilities and genericity, we believe one could implement a (1) *simple* to use, (2) *flexible*, (3) *type safe*, and (4) *performant* TPS library

without modifying the language, and, as discussed in this paper, also alternative abstractions for distributed interaction such as tuple spaces and RMI.

Pointing out the very fact that, to be extensible, an object-oriented language should be generic and reflective is not new (e.g., [38]). In this paper we have identified a precise case for this argument in the area of distributed computing. We have illustrated how our case poses more stringent demands than those previously expressed and partially addressed without distribution in mind, and have also more precisely quantified "how much" is missing in a current mainstream language such as Java. We insist on the fact that, in the face of modern abstractions for distributed interaction such as TPS, genericity needs to be provided in a form that includes runtime support for type parameters, and that reflection has to go beyond simple message reification (considered sufficient in the context of RMI, e.g., [3]). We pointed out the very fact that the current support in Java for genericity and reflection, from our perspective, is clearly insufficient.

Note that very few languages currently satisfy all our requirements. A reasonable candidate is Funnel [32], whose sole lack with respect to TPS is currently being repaired. Indeed, it is planned to instrument the Funnel language with a form of closures enabling a deferred evaluation. These will enable the expression of subscription patterns in a way conforming to the requirements posed by TPS.

## Acknowledgements

We would like to express our deepest gratitude to Gilad Bracha, Martin Odersky and Ole Lehrmann Madsen for highly valuable comments and suggestions on this work.

## References

1. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., and Chandra, T.D.: Matching Events in a Content-based Subscription System. In *Proceedings of ACM PODC99*, pp. 53-61, May 1999
2. Altherr, M., Erzberger, M., and Maffeis, S.: iBus - A Software Bus Middleware for the Java Platform. In *Proceedings of the Workshop on Reliable Middleware Systems of IEEE SRDS'99*, pp. 43-53, Octobre 1999.
3. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. In *Proceedings of ECOOP'93*, pp. 152-184, July, 1993.
4. Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM OOPSLA'98*, pp. 183-200, October 1998.
5. Brinch Hansen, P.: The Programming Language Concurrent Pascal. In *IEEE Transactions on Software Engineering*, 1(2), pp. 199-207, June 1975.
6. Bruce, K.B., Petersen, L., and Fiech, A.: *Subtyping Is Not a Good "Match" for Object-Oriented Languages*. In *Proceedings of ECOOP'97*, pp 104-127, June 1997.
7. Bruce, K.B., Schuett, A., and van Gent, R.: *PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language*, In *Proceedings of ECOOP'95*, pp 27-51, August 1995.

8. Bruun Kristensen, B., Lehrmann Madsen, O., Moller-Pedersen, B., and Nygaard, K.: Abstraction Mechanisms in the BETA Programming Language. In *Conference Record of ACM POPL'93*, pp. 285-289, January 1983.
9. Businger, A.: PORTAL Language Description. In *Lecture Notes in Computer Science*, number 198, Springer, 1985.
10. Canning, P., Cook, W.R., Hill, W., Olthoff, W., Mitchell, J.C.: F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings ACM FPCA'89*, pp 273-280, September 1989.
11. Cardelli, L.: A Language with Distributed Scope. In *Proceedings of ACM PODC'95*, pp. 286-297, August 1995.
12. Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G.: The Modula-3 Type System. In *Conference Record of ACM POPL'89*, pp. 202-212, January 1989.
13. Carzaniga, A., Rosenblum, D.S., and Wolf, A.L.: Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of ACM PODC 2000*, pp. 219-227, August, 2000.
14. Ciancarini, P. and Rossi, D.: *Jada: A Coordination Toolkit for Java*. ESPRIT Project #20197 – PageSpace.
15. Cugola, G., Nitto, E.D., and Fuggetta, A.: Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of ICSE'98*, pp. 261-270, April 1998.
16. Eugster, P.T. and Guerraoui, R.: Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of Usenix COOTS 2001*, pp. 131-146, January 2001.
17. Eugster, P.T., Guerraoui, R, and Damm, C.H.: On Objects and Events. In *Proceedings of ACM OOPSLA 2001*, pp. 131-146, October 2001.
18. Eugster, P.T., Guerraoui, R., and Sventek, J.: Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of ECOOP 2000*, Springer-Verlag, pp. 252-276, June 2000.
19. Eugster, P.T., Guerraoui, R., and Sventek, J.: Type-Based Publish/Subscribe. Technical Report DSC 200029, EPFL.
20. Freeman, E., Hupfer, S., and Arnold, K.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley, June 1999.
21. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
22. Gelernter, D.: Generative Communication in Linda. In *ACM TOPLAS*, 7(1), pp. 80-112, January 1985.
23. Gelernter, D., and Carriero, N.: Coordination languages and their significance. In *Communications of the ACM*, 35(2), pp. 97-107, February 1992.
24. Gosling, J., Joy, B., Steele, G., and Bracha, G.: The Java Language Specification, Second Edition. Addison-Wesley, 2000.
25. Happner, M., Burridge, R., and Sharma, R.: Java Message Service. Technical report, Sun Microsystems Inc., October 1998.
26. Haahr, M., Meier, R., Nixon, P., Cahill, V., and Jul, E.: Filtering and Scalability in the ECO Distributed Event Model. In *Proceedings of IEEE PDSE 2000*, pp. 83-92, 2000.
27. International Organization for Standardization: *Ada 95 Reference Manual - The Language - The Standard Libraries*. ANSI/ISO/IEC-8652:1995, January 1995.
28. Liskov, B.: Distributed Programming in Argus. In *Communications of the ACM*, 31(3), pp. 300-213, March 1988.
29. Liskov, B.: A History of CLU. In *ACM SIGPLAN Notices*, 28(3), pp. 133-147, March 1993.

30. Liskov, B. and Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of ACM SIGPLAN'88*, pp. 260-267, June 22-24, 1988.
31. Moussa, H.: Benchmarking Type-Based Publish/Subscribe. Project Report, Swiss Federal Institute of Technology, August 2001.
32. Odersky, M.: Functional Nets. In *Proceedings of the European Symposium on Programming*, pp. 1-25, March 2000.
33. OMG: *Notification Service Standalone Document*. OMG, June 2000.
34. OMG: *Common Object Services Specification, Chapter 4: Event Service*. OMG, March 2001.
35. Reiser, M.: The Oberon System, ACM Press, 1991.
36. Rivard, F.: Smalltalk : a Reflective Language. In *Proceedings of ACM Reflection'96)*, pages 21-38, April 1996.
37. Solorzano, J.H., and Alagic, S.: Parametric Polymorphism for Java: A Reflective Solution. In *Proceedings of ACM OOPSLA'98*, pages 216–225, October, 1998.
38. Steele, G.L.: Growing a Language. In *Higher-Order and Symbolic Computation* 12(3), pp 221-236, October 1999.
39. Stroustrup, B.: The C++ Programming Language, Third Edition. Addison-Wesley, June 1997.
40. Sun Microsystems Inc.: Java Remote Method Invocation - Distributed Computing for Java (White Paper), 1999.
41. Taha, W., and Sheard, T.: Multi-Stage Programming. In *Proceedings of ACM ICFP'97*, pp. 321–321, June, 1997.
42. Talarian Corporation: *Everything You Need To Know About Middleware*. White paper, `http://www.talarian.com`, 1999.
43. TIBCO Inc.: *TIB/Rendezvous White Paper*. White paper, `http://www.rv.-tibco.com/`, 1999.
44. Zenger, M., and Odersky, M.: Implementing Extensible Compilers. In *ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, June 2001.