

Linguistic Support for Distributed Programming Abstractions

Christian Heide Damm¹, Patrick Thomas Eugster^{2,*}, Rachid Guerraoui²

¹Microsoft Business Solutions, DK-2950 Vedb, Denmark

²Swiss Federal Institute of Technology in Lausanne, CH-1015 Lausanne, Switzerland

Abstract— *What abstractions are useful for distributed programming? This question has constituted an active area of research in the last decades and several candidate abstractions have been proposed, including remote method invocation, tuple spaces and publish/subscribe. How should such abstractions be offered to the programmer? Should they sit besides centralized programming abstractions in the core of a language? Should they rather sit within external libraries? Should they benefit from specific compiler support? These questions are also important but have sparked less enthusiasm.*

This paper contributes to addressing these questions in the context of Java and the type-based publish/subscribe (TPS) abstraction: an object-oriented variant of the publish/subscribe paradigm. We present an experience that compares implementations of TPS in (1) a variant of Java we designed to inherently support TPS, (2) standard Java, and (3) Java augmented with genericity.

We derive from our implementation experience general observations on what features a programming language should support in order to enable a satisfactory library implementation of TPS. In particular, we (re-) insist here on the importance of providing genericity and reflective features in the language, and point out the very fact that current efforts towards providing such features are still insufficient. We also argue that many of our observations do not only apply to TPS, but would also apply to alternative abstractions.

I. INTRODUCTION

When designing and implementing a distributed middleware [1], one of the first questions to address is *which* abstraction to provide to the programmer. Typical answers to this question are *remote procedure call* or *publish/subscribe*. A second, complementary, question then is *how* to implement the abstraction within the middleware.

One common technique, particularly employed in single-language academic settings, consists in an *integration* of the distributed programming abstraction with the programming language through specific primitives. That is, the distributed programming abstractions sit in the language, as first class citizens, besides traditional centralized abstractions (e.g., [2]). This approach might be motivated by performance and type safety, but is less adequate in the context of distributed programming. Indeed, the risk that the addition of primitives to a language might namely hamper portability and flexibility of that language is even more a concern in a distributed setting since it is still not clear which are the relevant abstractions.

A second approach is to rely on *compilation* for generating the glue between the abstraction (i.e., the middleware)

and the applications relying on it. The second approach can also provide type safety, and as shown by the success of RPC, seems to yield appealing results for programmers (e.g., [3]).

As illustrated by efforts in the Java community, which culminated in the introduction of the *dynamic proxy* as a mechanism supporting *remote method invocations* (RMI) without specific (pre)compilation, simple and general language features can offer very good support for implementing specific abstractions, e.g., for distributed programming, in a type-safe and elegant manner. A third approach to implementing specific abstractions for distributed programming consists in implementing those abstractions with no compilation support, in a type-safe and elegant manner, on top of the language by using such simple and general language features.

The motivation of this work was to figure out whether such an approach can be adopted for alternative abstractions to RMI, and more precisely for the *type-based publish/subscribe* (TPS) abstraction [4]. TPS is a variant of the publish/subscribe interaction scheme, and this scheme has shown to constitute a preferable alternative to RPC for certain kinds of applications. Roughly speaking, TPS is to publish/subscribe what remote method invocations are to the remote procedure calls: namely, an object-oriented variant of the paradigm. 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).

This paper compares three implementations of TPS. The first implementation is based on Java_{PS}, which is a variant of Java that we devised with specific primitives for supporting the TPS interaction style [4]. The second implementation [5] is based on standard Java. The third implementation is based on Generic Java (GJ) [6], 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.

Through this comparison, we point out how inherent reflective and generic capabilities could enable a satisfactory library implementation of TPS, refraining from any lan-

*Contact author. patrick.eugster@epfl.ch

guage extensions. While the importance of these capabilities has already been pointed out in other contexts, this paper argues, through TPS and Java, that current support of the capabilities in mainstream languages are still not sufficient for distributed computing.

Roadmap. The rest of the paper is organised as follows. Section II briefly overviews the TPS paradigm. Section III contains a short introduction to the three implementations of TPS. Sections IV-VII examine the approaches according to the four above-mentioned aspects. Section VIII summarizes the comparison and discusses a selected design alternative, and alternatives to Java. IX discusses related work on distributed programming abstractions, focusing on Java. Section X concludes the paper.

II. TYPE-BASED PUBLISH/SUBSCRIBE

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

A. Background: 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 [7], SmartSockets [8], iBus [9]). 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 large family of standards, e.g., Java Message Service (JMS) [10], CORBA Event & Notification Services [11], [12], or even JavaSpaces [13], 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 [14], Gryphon [15], Jedi [16]) 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

B. Overview of Type-Based Publish/Subscribe

Type-based publish/subscribe [4] (TPS) is a recent object-oriented variant of the publish/subscribe interaction style. In TPS, publishers 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 fields *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.

TPS is general, in the sense that it can be used to implement the traditional *content-based* publish/subscribe (e.g., [17], [18]), and hence also *subject-based* publish/subscribe

(e.g., [7], [8]). In a single-language setting, TPS can exploit the type system of the language at hand. TPS can, however, also be put to work in a heterogeneous environment [19].

C. 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, yet allows for optimisations when applying these filters. Clearly, TPS mainly aims at ensuring [4] (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, i.e., the underlying communication infrastructure must be granted insight into subscriptions. 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.

D. 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.

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 of companies with a name containing “Telco”, and whose price is less than 100.

III. THREE IMPLEMENTATIONS

This section gives a short introduction to the three implementations of TPS that we have considered. The first approach augments Java with primitives for TPS, resulting

```

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

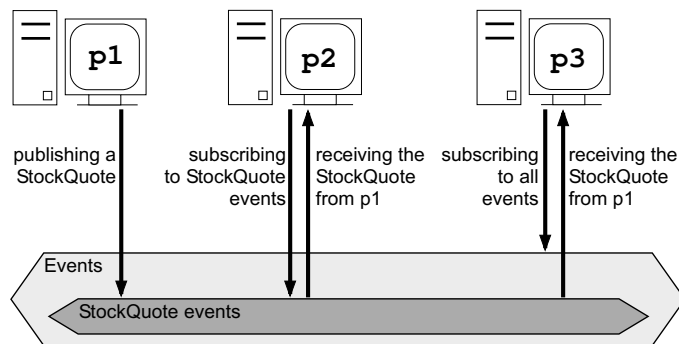


Fig. 2. Type-based publish/subscribe

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.

A. *Java_{PS}* Implementation

Java_{PS} [4] is a dialect of Java designed to support TPS through specific primitives:

```

publish Expression;
subscribe (EventType Identifier) Block Block;

```

The `publish` primitive publishes an event. The `subscribe` primitive generates a subscription to an event type. The first block represents a content filter referring to the actual event through an identifier, and the second block represents an event handler which is executed every time an event passes the filter and uses the same identifier. The `subscribe` primitive returns an expression of type `Subscription`, representing a handle for that subscription. Publishing a stock quote boils down to the following:

```

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

```

Subscribing to stock quotes can be expressed as follows:

```

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

```

Subscription s = subscribe (StockQuote q)
{
    return (q.getPrice() < 100 &&
           q.getCompany().indexOf("Telco") != -1);
}
{
    System.out.println("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. Note that the content filter is expressed in Java with the exact same code as in Section II-D above.

B. *Java Implementation*

Our Java implementation described in this section is based on our *Distributed Asynchronous Collections* (DACs) [5]. DACs are abstractions of object containers (e.g., a DAC can be queried with the `contains(Object)` method), which however differ from conventional collections by being asynchronous and essentially distributed. A DAC is thus not centralized on a single host, and operations may be invoked on it through local proxies from various nodes of a network. A DAC may also be used in an asynchronous way; 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).

Expressing one's 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. A subscription to an

```

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

Condition telcoCondition =
    new Equals("getCompany.indexOf",
        new Object[]{"Telco"},
        new Integer(-1));
Condition priceCondition =
    new Compare(".getPrice",
        new Object[]{new Integer(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

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.

Figure 4 illustrates how a stock broker issues a subscription through a DAC representing type `StockQuote` (the instantiated DAC class `DAS` [5] 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 VII).

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("Telco0ps", 80, 10);
stockQuotes.add(q);

```

C. GJ Implementation

In the previously described Java implementation of TPS, 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.

The generic library approach is based on GJ [6], which is an extension of Java with support for genericity through

```

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

parametric polymorphism. 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 `GSubscriber`, there is no longer a need for a subject name parameter in the callback method.

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("Telco0ps", 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.

IV. 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.

A. Content Filters

In our `JavaPS` implementation, the content filters are truly expressed in the programming language at hand, making them simple to express for programmers familiar

with that 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` [4].

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. Unfortunately, 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.

B. Qualities of Service

The limited form of QoS expressed through the specific (G)DAC type, e.g., (G)DASet for reliable communication (see [5]), enables the use of the same event types with different and maybe even incompatible QoS: a publisher can publish events of a given type through a (G)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. With the current (G)DAC implementations, developers are expected to ensure manually that (G)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 `JavaPS` implementation by expressing the QoS through the events themselves. QoS are associated with event types, which are in fact the only “contract” between publishers and subscribers.

C. 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 `JavaPS` 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.

Verdict: Our TPS-specific language primitives in `JavaPS` offer a very concise syntax: subscription expressions are compact and use a subset of native Java syntax, which makes them easily understandable. The Java and GJ implementations both suffer from possible mismatches in QoS. In addition, filter expression in these two approaches suffers from a heavy syntax, and in particular from the lack of custom **operator overloading** inherent to Java when combining simple conditions.

V. 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.

This aspect is important, because an implementation of publish/subscribe which is very specific, and hence limited, can quite easily provide good simplicity and readability. pretty obvious that one comparison without the others doesn’t make sense

A. Content Filters

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 optimizations.

In our `JavaPS` 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.

B. Qualities of Service

In our `JavaPS` 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.

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 `JavaPS` 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.

Verdict: A library will always be more flexible than a solution integrated in the language, since the latter type of solution is more tedious to modify. 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 than `JavaPS`.

VI. 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.

We compare here how the different implementations ensure type safety, one of the two main driving forces behind TPS.¹ Obviously, the potential level of type safety that can be achieved depends on the considered language itself, and mechanisms such as reflection, can be misused to willingly introduce type errors.

A. 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 receive.

In our `JavaPS` 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 III), which can lead to possible mismatches.

B. Content Filters

The content filters in our `JavaPS` 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 III-B), through the **introspection** capabilities of Java.

Note, however, that the developer, though not using reflection explicitly to define *which* methods (and arguments)

¹Encapsulation, as the second main motivation for TPS, is ensured in all three implementations through the programming model promoted by TPS.

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 methods.

Verdict: Not surprisingly, type safety increases in the GJ implementation compared to the Java implementation, and increases further with `JavaPS`, where there can be no “type unsafety” related to TPS.

The GJ implementation ensures type safety when publishing and receiving events, yet can not provide such guarantees for content filters. In latter context, type safety would however be more important, as Java programmers are used to untyped collections.

VII. PERFORMANCE

Last but not least, we present the most significant results of our performance measurements realized with the three different approaches. We actually measure the overhead of the GJ and Java approaches with respect to `JavaPS`.

A. Setting

We have used the same simple architecture as testbed for all three implementations. 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 is essentially the same in all three approaches, we have focused on type-based filtering.

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 (perceived by the publisher) onto the corresponding channel. [20] provides information on further measures.

B. Library vs Language Integration

The two library implementations differ from the implementation of `JavaPS`, in that upon publishing an event, the precise channel for the corresponding class has to be found. In the case of `JavaPS`, 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 `JavaPS` 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.

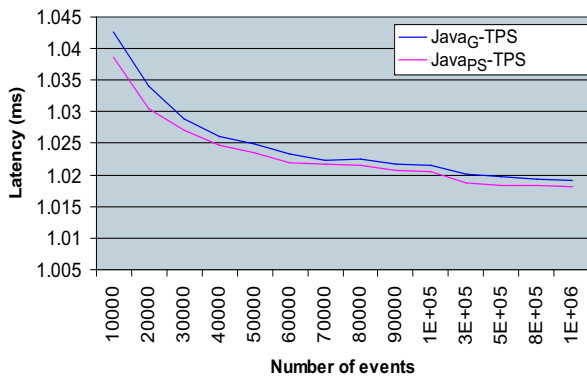
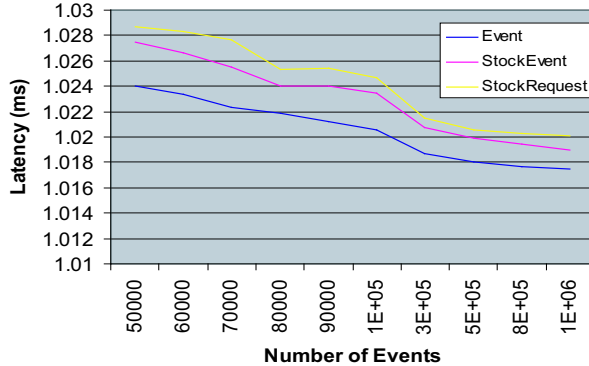
Fig. 6. Latency of publishing: Java_{PS} vs GJ

Fig. 7. Latency of publishing different event types

C. 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 measurements 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.

Verdict: The latency observed when publishing events is slightly, but clearly, smaller in the case of Java_{PS} than with the Java or GJ implementations. This 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. (Optimisations for the involved channel lookups could certainly be performed.)

This section first presents a summary of the how the three implementations perform with respect to the chosen comparison aspects. Then, an alternative programming language mechanism for improving the library implementation(s) is presented, and finally, the main language mechanisms pointed out so far are investigated in the context of .NET [21].

	Java	GJ	Java _{PS}
Simplicity	~	~	+
Flexibility	+	+	÷
Type safety	÷	~	+
Performance	~	~	+

TABLE I

COMPARISON SUMMARY (÷ INSUFFICIENT, ~ ACCEPTABLE, + GOOD)

A. Summary

Table I 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.

This is not fully surprising, as Java_{PS} was motivated by the obvious lacks manifested by the Java language with respect to TPS, after some of those lacks had already been addressed by using a “future” version of Java incorporating genericity.

B. Dynamic Proxies

Especially for the library implementations of TPS there are many alternative design choices, and many tradeoffs involved (see [20]). The weakest point of both these approaches, as mentioned above, is related to the unwieldy content filter expression. Dynamic proxies, a simple mechanism for **behavioral reflection** in Java, can improve type safety in filter expression. For instance, the asynchronous `contains()` method in DACs can be modified to return a dynamic proxy which “registers” the invocations performed on it:

```
GDAC<StockQuote> stockQuotes = ...;
StockQuote q = stockQuotes.contains(...);
q.getCompany().equals("TelcoOps");
```

The expression of interest in stock quotes of a given company through a proxy *q* reveals however the weaknesses of dynamic proxies. Only strict equality can be expressed, and attributes of primitive types can not be matched. Indeed, as operators such as `>` or also `!=` are not reified as method invocations (this would come with operator overloading, see Section IV). Furthermore, the above code would fail at runtime, as dynamic proxies can only be created for interface types.

C. .NET

Definitely inspired by the Java technology, Microsoft's .NET [21] platform lends itself well to a side-by-side comparison. Being of a more recent date, it is interesting to investigate whether .NET qualifies better for distribution than Java, in particular in supporting demanding abstractions like TPS.

From a general point of view, .NET does indeed provide many mechanisms for distribution, with slightly more variants than Java. Alone for serializing objects, .NET provides three mechanisms (binary, SOAP, custom). .NET *remoting*, the .NET counterpart to Java RMI, provides many means of configuring remote invocations. With respect to the main mechanisms discussed so far, which are *genericity*, *reflection*, and *types*, .NET can be summarized as follows: *Genericity*: Genericity has been investigated in the context of .NET [22]. It is however not currently integrated, but rather "foreseen for a future release".

Reflection: .NET provides roughly the same introspection features as Java, and furthermore also provides dynamic proxies, however similarly restricted to interface types.

Types: Remotely invocable objects in .NET can also be of class types, and direct field accesses are automatically transformed to access method invocations. Furthermore, .NET also offers support for transformations between primitive types and wrapper types, referred to as *boxing* and *unboxing* respectively.

In other terms, .NET makes a step more towards the ideal language for TPS, by enforcing encapsulation. Unfortunately however, the state of genericity is not as advanced in .NET as it is in Java (though genericity is still not finalized there either).

IX. 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 [23]. In that context however, the question of what language mechanisms would help avoiding any extension is devoted less attention. In the following, we hence 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.

A. Tuple Spaces

The TS abstraction first appeared in the Linda programming language [24], in which spaces served as coordination means between cooperating processes.

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;
in ("StockQuote", "Telco", i, 10); //2
in ("StockQuote", "Telco", var i, 10); //3
in ("StockQuote", "Telco", j: integer, 10); //4
```

In line 1, a tuple consisting of 4 values is put into the tuple space. In line 2, 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 3 causes the `i` to be treated as a formal parameter, i.e., it can match any value. In line 4, the `integer` keyword at the same time declares a new variable `j` and uses it as a formal parameter as in line 6.

Implementing TSs in Java poses similar problems to those we described for TPS in this paper. As an example, Jada [25] 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 4 has no equivalent in Jada):

```
Integer k = new Integer(10); //for brevity
TupleSpace tupleSpace = new TupleSpace();
tupleSpace.out(new Tuple("StockQuote", "Telco",
    new Integer(80), k)); //1
int i = 80;
Tuple tuple1 = tupleSpace.in(new Tuple(
    "StockQuote", "Telco", new Integer(i), k)); //2
Tuple tuple2 = tupleSpace.in(new Tuple(
    "StockQuote", "Telco", Integer.class, k)); //3
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` [13], 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.

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 [26], 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.

B. RMI

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

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 `RemoteExceptions`. 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.

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 discussed in Section VIII-B 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 VIII-B. In the case of RMI, it is not clear whether this mechanism for behavioural reflection should and 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.

X. CONCLUSIONS

In the face of today’s heterogeneity across platforms, we believe that designers of future languages should foresee a general support for distributed programming abstractions. This is particularly important for not to commit on a limited set of abstractions but rather leave the possibility of adding new ones according to the programmer’s needs.

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. We argue that reflection, just like genericity, as faces of *extensibility*, are the key concepts for a general language support of distributed

programming. With inherent and uniform reflective capabilities and genericity, we believe one could implement a (1) *simple* to use, (2) *flexible*, (3) *type safe*, and (4) *performant* TPS library in the language itself, and also alternative abstractions for distributed programming such as tuple spaces and RMI (see [20]).

Pointing out the very fact that, to be extensible, an object-oriented language should be generic and reflective is not new (e.g., [27]). In this paper we have identified a precise case for this argument in the area of distributed computing, and illustrated how our case poses more stringent demands than those previously expressed and partially addressed without distribution in mind. We insist on the fact that, in the face of modern abstractions for distributed programming 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., [28]). We pointed out the very fact that the current support in Java for genericity and reflection, from our perspective, is clearly insufficient. Unfortunately, it seems that by inheriting concepts from Java, Microsoft’s .NET platform also inherited most of the corresponding weaknesses.

ACKNOWLEDGEMENTS

We would like to express our deepest gratitude to Gilad Bracha, Martin Odersky and Ole Lehrmann Madsen for highly valuable comments and suggestions, which have helped improving this paper.

REFERENCES

- [1] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Associates, 2000.
- [2] B. Liskov and R. Sheifler, “Guardians and Actions: Linguistic Support for Robust, Distributed Programs,” in *Conference Record of the 9th ACM Symposium on Principles of Programming Languages (POPL ’82)*, 1982.
- [3] OMG, *The Common Object Request Broker: Architecture and Specification*, OMG, February 2001.
- [4] P.Th. Eugster, R. Guerraoui, and C.H. Damm, “On Objects and Events,” in *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, Oct. 2001, pp. 131–146.
- [5] P.Th. 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, pp. 252–276.
- [6] 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)*, Oct. 1998, pp. 183–200.
- [7] TIBCO, *TIB/Rendezvous White Paper*, <http://www.rv.tibco.com/>, 1999.
- [8] Talarian Corporation, *Everything You need to Know about Middleware: Mission-Critical Interprocess Communication (White Paper)*, <http://www.talarian.com/>, 1999.
- [9] M. Altherr, M. Erzberger, and S. Maffei, “iBus - A Software Bus Middleware for the Java Platform,” in *Proceedings of the International Workshop on Reliable Middleware Systems (SRDS ’99)*, Oct. 1999, pp. 43–53.
- [10] M. Happner, R. Burridge, and R. Sharma, “Java Message Service,” Tech. Rep., Sun Microsystems Inc., Oct. 1998.

- [11] OMG, *CORBA services: Common Object Services Specification, Chapter 4: Event Service*, OMG, March 2001.
- [12] OMG, *Notification Service Standalone Document*, OMG, June 2000.
- [13] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley, June 1999.
- [14] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service," in *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, July 2000, pp. 219–227.
- [15] 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)*, Nov. 1999, pp. 53–62.
- [16] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems," in *Proceedings of the 10th IEEE International Conference on Software Engineering (ICSE '98)*, Apr. 1998, pp. 261–270.
- [17] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, vol. 33, no. 3, pp. 68–76, Mar. 2000.
- [18] G. Mühl, L. Fiege, and A.P. Buchmann, "Filter Similarities in Content-Based Publish/Subscribe Systems," in *Proceedings of the 2002 International Conference on Architecture of Computing Systems (ARCS 2002)*, Apr. 2002, pp. 224–240.
- [19] S. Baehni, P.Th. Eugster, R. Guerraoui, and P. Altherr, "Pragmatic Type Interoperability," in *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS '03)*, May 2003.
- [20] C.H. Damm, P.Th. Eugster, and R. Guerraoui, "Abstractions for Distributed Interaction: Guests or Relatives?," Tech. Rep. DSC/2001/052, Swiss Federal Institute of Technology in Lausanne, June 2000.
- [21] H. Lam Th. Thai, *.NET Framework Essentials*, O'Reilly and Associates, Inc., June 2001.
- [22] A. Kennedy and D. Syme, "Design and Implementation of Generics for the .NET Common Language Runtime," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001.
- [23] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul, "Filtering and Scalability in the ECO Distributed Event Model," in *Proceedings of the 5th IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, June 2000, pp. 83–92.
- [24] D. Gelernter, "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.
- [25] P. Ciancarini and D. Rossi, "Jada - Coordination and Communication for Java Agents," in *Mobile Object Systems: Towards the Programmable Internet*, vol. 1222 of *LNCS*, pp. 213–228. Springer, Apr. 1997.
- [26] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.
- [27] G.L. Steele, "Growing a language," *Higher-Order and Symbolic Computation*, vol. 12, no. 3, pp. 221–236, Oct. 1999.
- [28] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, "Abstracting Object Interactions Using Composition Filters," in *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, July 1993, pp. 152–184.