Linguistic Support for Distributed Programming Abstractions*

Christian Heide Damm
Microsoft Business Solutions
DK-2950 Vedb, Denmark

Patrick Thomas Eugster†
Sun Microsystems
CH-8604 Volketswil, Switzerland

Rachid Guerraoui
Distr. Progr. Laboratory, EPFL
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 invocations, 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 enthousiasm.*

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, and finally, also alternative abstractions 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.

1. Introduction

When designing and implementing a distributed middleware, one of the first questions to address is *which* abstraction to provide to the programer. 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., [8]). This approach might be motivated by performance and type safety, but might hamper portability and flexibility of the considered programming language, which is a strong concern in distributed settings 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. This approach can also provide type safety, and as shown by the success of RPC/RMI, seems to yield appealing results for programmers (cf. CORBA).

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 precisely in implementing those abstractions in a type-safe manner by using only such simple and general language features.

The motivation of this work was to find out whether such an approach can be adopted for alternative abstractions to RMI, and more precisely for the *type-based publish/subscribe* (TPS) abstraction [6], a variant of the publish/subscribe interaction scheme. Roughly speaking, TPS is to publish/subscribe what the RMI is to the RPC: namely, an object-oriented variant of the paradigm. Just like RMI, 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

* Financially supported by the Swiss National Science Foundation - NCCR / PRN MICS IP 5.2

† Contact author. Former affilation: Distributed Programming Laboratory, EPFL, patrick.eugster@epfl.ch

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 [6]. The second implementation [7] 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 Java at version 1.5).

We consider four comparison axes: (1) *simplicity*, (2) *flexibility*, (3) *type safety*, and (4) *performance*. Through this comparison, we point out how inherent reflective and generic capabilities could enable a satisfactory library implementation of TPS, refraining from any language 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 organized as follows. Section 2 briefly overviews the TPS paradigm. Section 3 contains a short introduction to the three implementations of TPS. Sections 4-7 examine the approaches according to the four above-mentioned aspects. Section 8 summarizes the comparison and discusses a selected design alternative. Section 9 concludes the paper.

2. 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.

2.1. Overview of type-based publish/subscribe

Type-based publish/subscribe [6] (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., [1]), and hence also *subject-based* publish/subscribe (e.g., [10]). 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 [3].

2.2. 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 optimizations when applying these filters. Clearly, TPS mainly aims at ensuring [6] (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, decentralized applications in which performance is a primary concern, (4) *open content filters* are important to enable optimizations 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) *Quality 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 these challenges.

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
```

3. 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 in a dialect of Java called Java$_{PS}$. The second approach is an

```
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;
  }
}
```

Figure 1. Simple stock quote events

Figure 2. Type-based publish/subscribe

implementation of TPS in standard Java, while the last approach is based on GJ, which adds *genericity* to Java.

3.1. Java$_{PS}$ implementation

Java$_{PS}$ [6] 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:

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

```
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);
}
```

Figure 3. API of the Java implementation

Note that the content filter is expressed in Java with the exact same code as in Section 2.3 above.

3.2. Java implementation

Our Java implementation described in this section is based on our *Distributed Asynchronous Collections* (DACs) [7]. 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 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. 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.

```
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);
```

Figure 4. Subscribing with DACs

Figure 4 illustrates how a stock broker issues a subscription through a DAC representing type `StockQuote` (the instantiated DAC class `DAS` [7] 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 optimization (see Section 7).

Similarly, the stock market publishes quotes through the DAC representing the type `StockQuote` like this:

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

3.3. 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 correct 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 [4], which is an extension of Java with support for genericity through *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.

```
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);
}
```

Figure 5. API of the GJ implementation

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("TelcoOps", 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.

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. Simplicity does not necessarily favor 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

In our Java$_{PS}$ 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` (see [6]).

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 `telcoCondition` 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. QoS

The limited form of QoS expressed through the specific (G)DAC type, e.g., (G)DAS for reliable communication (see [7]), 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 Java$_{PS}$ 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.

4.3. 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.

Verdict: Our TPS-specific language primitives in Java$_{PS}$ 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.

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 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.

5.1. 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 such custom conditions must provide several hooks in order to nevertheless enforce optimizitions.

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. QoS

In our Java$_{PS}$ implementation, the QoS 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 re-

flect the guarantees they offer. In our Java$_{PS}$ implementation, such a customization 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 Java$_{PS}$.

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.

We compare here how the different implementations ensure type safety, one of the two main driving forces behind TPS. 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.

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 receive.

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.

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.

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 methods.

Verdict: Not surprisingly, type safety increases in the GJ implementation compared to the Java implementation, and increases further with Java$_{PS}$, 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.

7. 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 Java$_{PS}$.

7.1. 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. [5] provides information on further measures.

Figure 6(a). Latency of publishing: Java$_{PS}$ vs GJ

Figure 6(b). Latency with event types

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(a), 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.

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 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 6(b) 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 `StockQuote` 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. (Optimizations are of course possible.)

8. Discussion

This section first presents a summary of how the three implementations perform with respect to the chosen comparison aspects, and then, presents an alternative programming language mechanism for improving the library implementation(s).

	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)

8.1. Summary

Table 1 summarizes 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.

8.2. Dynamic proxies

Especially for the library implementations of TPS there are many alternative design choices, and many tradeoffs involved (see [5]). 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* reveils 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 4). Furthermore, the above code would fail at runtime, as dynamic proxies can only be created for interfaces.

9. Conclusions

In the face of today's heterogenity across platforms, we believe that designers of future languages should foresee a general support for distributed programming abstractions.

Although TPS is surely not the last paradigm for distributed programming, the constraints imposed by TPS should be kept in mind when conceiving such support. 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 [5]).

Pointing out the very fact that, to be extensible, an object-oriented language should be generic and reflective is not new (e.g., [9]). 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., [2]). We pointed out the very fact that the current support in Java for genericity and reflection, from our perspective, is clearly insufficient.

Acknowledgements

We are very grateful to Gilad Bracha, Martin Odersky, and Ole Lehrmann Madsen for commenting on an earlier version of this paper.

References

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–62, Nov. 1999.

[2] 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)*, pages 152–184, July 1993.

[3] S. Baehni, P. 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.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 183–200, Oct. 1998.

[5] C. Damm, P. Eugster, and R. Guerraoui. Abstractions for Distributed Interaction: Guests or Relatives? Technical Report DSC/2001/052, Swiss Federal Institute of Technology in Lausanne, June 2000.

[6] P. Eugster, R. Guerraoui, and C. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, Oct. 2001.

[7] P. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[8] 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.

[9] G. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, Oct. 1999.

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