Towards Safe Distributed Application Development*

Patrick Thomas Eugster†
Sun Microsystems
CH-8604 Volketswil, Switzerland
patrick.eugster@sun.com

Christian Heide Damm
Microsoft Business Solutions
DK-2950 Vedb, Denmark
chrisd@microsoft.com

Rachid Guerraoui
Distr. Progr. Laboratory, EPFL
CH-1015 Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Abstract

Distributed application development is overly tedious, as the dynamic composition of distributed components is hard to combine with static safety with respect to types (type safety) and data (encapsulation). Achieving such safety usually goes through specific compilation to generate the glue between components, or making use of a single programming language for all individual components with a hardwired abstraction for the distributed interaction.

In this paper, we investigate general-purpose programming language features for supporting third-party implementations of programming abstractions for distributed interaction among components. We report from our experiences in developing a stock market application based on type-based publish/subscribe (TPS) implemented (1) as a library in standard Java as well as with (2) a homegrown extension of the Java language augmented with specific primitives for TPS, motivated by the lacks of former implementation. We then revisit the library approach, investigating the impact of genericity, reflective features, and the type system, on the implementation of a satisfactory TPS library. We then discuss the impact of these features also on other distributed programming abstractions, and hence on the engineering of distributed applications in general, pointing out lacks of mainstream programming environments such as Java as well as .NET.

1 Introduction

The engineering of industrial-scale *distributed* applications, is still overly tedious, and in particular, intrinsically harder than that of applications targeted at a deployment on a single host [11, 17]. The development of distributed applications not only leads to dealing with the the application logic (e.g., data types, semantics) itself, but furthermore with the remote interaction between distributed components taking place at runtime.

Since, at the exception of a rapidly decreasing proportion, currently developed industrial-scale applications all involve several physically distributed components, it is of paramount importance to come up with solutions for connecting distributed components in a way providing *safety*. This includes safety in the sense of (1) the distributed interaction (*safety properties* in the face of network node and link failures) taking place at execution, but also in the sense of (2) data types (*type safety*), and (3) data state (*encapsulation*). While safety in the sense of (1) depends on the abstraction provided by a middleware system (e.g., *remote procedure call* (RPC), *tuple space*, *publish/subscribe*) and the distributed algorithms chosen to implement the remote interaction underlying that abstraction in a given system model, safety in the sense of (2) and (3) are more a consequence of how the chosen abstraction, and actually the middleware implementing it, are put to work.

As pointed out in [20], there is a need for various abstractions, and thus potentially different middleware solutions, also because achieving safety in the sense of (1) in a given application context potentially requires a specific abstraction. Hence, a straightforward solution consisting in hardwiring an abstraction (or several ones) into a programming language in order to achieve (2) and (3) tends to make the language overly complex. A concept able of capturing different abstractions, and hence various middlewares and architectural styles [10] are *connectors*. From an architectural perspective, connectors make the composition of (physically distributed) components easier, by playing the role of glue between the different components. It is then precisely at the interface to these connectors that safety in the sense of (2) and (3) is difficult to obtain.

The goal of this paper is to investigate programming language support for the implementation of connectors as "libraries".[1] Of interest are solutions relying solely on pro-

*Partially funded by Lombard Odier Darier Hentsch Co..

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

[1] In this paper, "library implementation" refers to an implementation in

gramming language features which are *inherent* (avoiding any specific explicit compilation) and *general*. This precludes not only specific support for connectors (e.g., [2]), but also for some particular (distributed) programming abstraction (e.g, [18]).

More precisely, we investigate *genericity* and *reflection*, through experience gathered by developing a stock market application [22] with *type-based publish/subscribe* (TPS) [12], a variant of the publish/subscribe abstraction [5, 8]. We first present two of our TPS implementations, namely (1) based on a library developed in standard Java (referred to as "Java-TPS") and (2) based on Java$_{PS}$, which is a variant of Java we devised with specific primitives for supporting the TPS interaction style ("Java$_{PS}$-TPS"). The goal is to illustrate the gap between current possibilities in a language such as Java, and an "optimal" solution such as Java$_{PS}$. Then, we present a novel "enhanced" library implementation ("Java$_{G}$-TPS"), exploiting recent and experimental features of Java such as Generic Java (GJ) [4], an extension of Java providing genericity.

Making use of reflection in the context of connectors is a common practice (e.g, [28]), and the benefits of genericity have already been pointed out (though mostly in the perspective of centralized applications, e.g., [23]). In this paper, we go a step further, by pinpointing lacks of current and future support of these features, but also of the type system, in a mainstream language such as Java. As we point out in this paper, TPS is demanding enough that its requirements with respect to the investigated language features cover those of many other abstractions for developing distributed applications, and connectors in general.

Roadmap. The rest of the paper is organized as follows. Section 2 briefly introduces the TPS paradigm, along with a sample application that we will be using throughout the paper. Section 3 presents overviews of Java-TPS and Java$_{PS}$-TPS. Sections 4-6 then discuss general-purpose language features investigated in the context of Java$_{G}$-TPS. Section 7 wraps issues such as performance considerations and alternative features. Section 8 discusses related abstractions for distributed programming, and Section 9 concludes the paper.

2 Type-based publish/subscribe (TPS)

2.1 Overview

Type-based publish/subscribe [12] (TPS) is a recent object-oriented instantiation 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 such objects. A subscription can furthermore have a *content filter* associated, which is basically a predicate expressed on the public members of the type, including fields *as well as* methods (unlike related forms of typed event programming, e.g., [15]). Since event objects are instances of application-defined types, they are first-class citizens.

TPS is general, in the sense that it can be used to implement traditional *content-based* publish/subscribe (e.g., [8, 5]), and hence also *subject-based* publish/subscribe (e.g., [27, 7]). 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 heterogenous environment, provided a common code representation ("byte code") is available [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 aims at ensuring [12] (1) *type-safety* and (2) *encapsulation* with (3) *application-defined event types*. Since TPS aims at large-scale, decentralized applications, (4) *open content filters* are important to enable optimizations in the underlying 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.

2.3 Running example: a TPS application

We describe below a simplified version of an application developed with TPS [22]. It is used throughout this paper to examine how our different 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 participant p1 publishes a stock quote, i.e., an instance of the type `StockQuote`. Participant p2 has subscribed to the `StockQuote` type and thus receives the stock quote published by p1. Participant p3 has subscribed to the `Event`

the considered programming language without any hooks into the runtime environment, i.e., implementable by a third party.

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 *any* 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$.

```
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

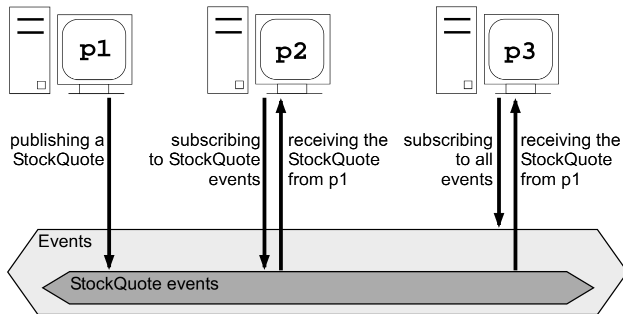

Figure 2: Interaction in the stock application

3 Two ways of developing with TPS

3.1 Java-TPS

Most related systems are implemented with language bindings to standard Java, promoting a first-class abstraction of a ubiquitous communication channel. Similarly, our Java implementation described in this section is based on our *Distributed Asynchronous Collections* (DACs) [14]. 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).

```
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: Core API for Java-TPS

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 implement TPS. A subscription to an event type (and implicitly, its subtypes) is issued through a DAC representing that type.

Figure 4 illustrates how a stock broker issues a subscription through a DAC representing type `StockQuote` (the instantiated DAC class `DAS` reflects reliable delivery [14]). The awkward appearance of the filter is motivated by the special requirements on content filters, such as their undergoing of deferred evaluation to enforce prior optimization. More precisely, filters are transformed at runtime into instantiations of first class abstract syntax trees. This makes these filters "migratable" and transformable. As mentioned in Section 2.2, this is done in order to offer the underlying TPS middleware an insight into these filters, and make optimizations possible when remotely evaluating them, e.g., partial evaluation only.

The stock market publishes stock quotes also through the DAC representing the type `StockQuote` like this:

```
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[]{100}, -1);
Condition contentFilter =
  telcoCondition.not().and(priceCondition);
Subscriber subscriber = new StockQuoteSubscriber();
DAC stockQuotes = new DAS("StockQuote");
stockQuotes.contains(subscriber, contentFilter);
```

Figure 4: Subscribing in Java-TPS

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

Besides type safety problems already known from centralized collections in Java (manifesting here through possible mismatches in types when casting events or setting up DACs), and those added by content filters (through the cumbersome explicit instantation of abstract syntax trees), also the scheme for expressing QoS can lead to mismatches in Java-TPS. A publisher can for instance produce events (e.g., stock quotes) with reliable semantics, while a subscriber can register for these same stock quotes but indicate the tolerance to losses, i.e, expressing unreliable semantics.

3.2 Java$_{PS}$-TPS

Motivated by the lacks of Java-TPS, Java$_{PS}$ [12] is a dialect of Java designed specifically to support TPS. Hence, Java$_{PS}$-TPS can be seen as a proposal for an *optimally* safe TPS, which integrates TPS by adding two new primitives to the original Java language:

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

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.

When developing with Java$_{PS}$-TPS, content filters are truly expressed in the native language, relieving the burden on developers already familiar with the language. More precisely, filters are implemented by a form of *deferred code evaluation* (similar, though not identical, in spirit to MetaML [25]) to ensure that these can be type-checked at compilation, yet without being compiled directly. There are, however, restrictions on what variables can be accessed inside content filters, to make these filters easily transferrable in a distributed environment (details can be found in [12]).

Using these primitives, a stock quote can be published like the following:

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

Subscribing to stock quotes occurs as follows:

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

Note that the content filter has the same semantics as the one expressed in Figure 4.

The obvious drawback of Java$_{PS}$-TPS is the proprietary extension of the Java language and the resulting requirement for a specific compiler.

4 Genericity

4.1 Java$_G$-TPS — generic DACs

As with any collections in current Java, the DACs promoted by Java-TPS mandate manual type casts of received events, and furthermore, type checks of inserted (published) events. Using **genericity**, these burdens can be avoided. Clearly, the event handler and the content filter associated with a subscription have to agree on the considered event type, and this correspondance can be sealed at the instantiation of the TPS abstraction by using genericity.

Since Java does not (currently) support genericity, the exploratory implementation of TPS, Java$_G$-TPS, is based on GJ [4]. With GJ, we obtain *typed* DACs without generating type-specific code, and nevertheless avoid explicit type casts. The resulting generic DACs (GDACs) and associated types are outlined in Figure 5. As a result of the type parameterized subscriber, there is no longer a need for a subject (i.e., type) name parameter in the callback method of subscribers (`GSubscriber`).

4.2 Developing with Java$_G$-TPS

Using this generic version of DACs, stock quotes can be published like this:

```
interface GDAC<T> {
  boolean add(T event);
  T get();
  boolean contains(T event);
  T contains(GSubscriber<T> subscriber);
  ...
}
interface GSubscriber<T> {
  void notify(T event);
}
```

Figure 5: Core API for Java$_G$-TPS

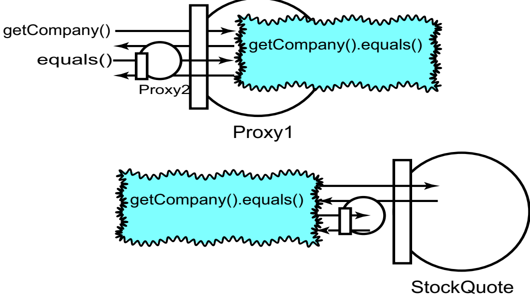

Figure 6: Content filters in Java$_G$-TPS

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

Note that the parameter passed to the GDAC constructor above is necessary, since the original implementation GJ used does not provide **runtime information on type parameters**. Clearly, although a GDAC is instantiated for a given type `StockQuote` in the example in Section 4.2, the constructor requires an explicit argument representing a reification of that type precisely. With runtime support, this somewhat redundant argument could be avoided. Luckily, it appears that the current proposition for adding generics to Java, now provides some information on type parameters. The content filter expression used in Java-TPS will be modified and illustrated in the next section.

4.3 Beyond TPS — statically type safe components

The benefits of genericity, roughly the ensuring of type safety in interactions between library classes and instances of (new) application-defined types, have already become apparent in centralized contexts. Collections, in particular, have been widely used to demonstrate the usefulness of genericity.

In applications with physically distributed components, genericity becomes even more important. The increased potential size of distributed applications, along with constraints such as 24×7 requirements makes it even more unfeasible to rewrite, regenerate, or recompile code in order to enforce type safety. Genericity, as a mechanism enforcing reuse of code, can fully develop its power in a distributed setting where late binding is required.

Genericity is hence not only useful in the context of TPS. A generic lookup service ("registry") could also improve type safety in RMI implementations, and help avoiding type checks and casts. This is particularly valid in Java RMI, since all remote interactions are statically typed, and hence the type of an obtained remote object reference has to be known anyhow (see Section 8.2).

5 Reflection

5.1 Content filter expression

The main remaining flaw from Java-TPS is its clearly unsatisfactory expression of content filters as pointed out in Section 3.1.

An alternative mechanism for the type-safe expression of content filters was anticipated with the integration of **behavioral reflection** with Java 1.3 through the `Proxy` class: while the introspection mechanisms provided in Java since version 1.1 enable the reification of methods and the dynamic invocation of these, behavioral reflection allows the interception of (also) statically typed method invocations, and the performing of any action in the confines of these invocations. Such a scheme could allow the developer to express queries on event objects by making the corresponding invocations on a proxy, which would *record* these (reified) invocations, such that they could be *replayed* on the effective events for filtering (see Figure 6). To that end, the `contains()` method of the (G)DAC interface would be changed, to return an instance of a proxy class bound to an `InvocationHandler`, which would register the invocations performed on the proxy.

Using this, we could express interest in all stock quotes from company "Telco" on a proxy `q` as follows (recorded invocations are *emphasized*):

```
class QuoteSubscriber
  implements GSubscriber<StockQuote> {...}
GDAC<StockQuote> qs =
  new GDAS<StockQuote>(StockQuote.class);
StockQuote q = qs.contains(new QuoteSubscriber());
q.getCompany().equals("Telco");
```

Although this approach can simplify content filter expression to some extent, it only supports relatively simple filters (without conditionals and loops etc.). In addition, the above example can not be made to work as such, since both the `StockQuote` and `String` types are not interfaces. This restriction is due to the current implementation of dynamic proxies (discussed further in the next section). Further limitations of Java account for the fact that the con-

dition expressed on the company name is stricter than the one used so far.

5.2 Behind the scene

Prior to its use in prototypical content filter expression, reflection has already been applied in "hidden", but nevertheless crucial points of our TPS implementation(s) (e.g., Section 3.1).

Given the nature of our distributed context, there must be a way for distributed components, 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**. The runtime support for type parameters pointed as in Section 4.2, can be viewed as such reflection support for genericity.

Reflection is also crucial for the implementation and handling of events. First, **serialization**, which in Java relies on introspection, is a very convenient means of converting event objects to conveyable network messages. Second, **dynamic class loading and linking** is another useful feature of Java which is often considered as being of reflection. Even if remote components agree on the same types of exchanged events, nothing should prevent them from introducing new conformant event classes at runtime.

5.3 Beyond TPS — safe dynamic composition

As discussed in Section 4, genericity enables the implementation of abstractions in a way providing type safe API's. In a centralized setting, this may suffice and even ensure type safety statically.

In a distributed context, however, it is very likely that runtime type checks have to be performed. Indeed, network channels and hence layers "close" to the network and corresponding libraries are inherently untyped. It appears hence natural that any implementation of a library for distributed programming will require explicit type inclusion checks performed dynamically (i.e., against types unknown at compilation).

While genericity can be seen as supporting safe interaction with a library implementation of an abstraction for distributed interaction, reflection can be seen as supporting a safe implementation of such an abstraction. Not only **reification** of **types** (cf. structural reflection), but also reification of **computation** (cf. behavioral reflection), though potentially associated with performance penalties, appear to be the best way to combine (1) statically safe interaction between individual components and the abstractions for distributed interaction they rely upon, and (2) dynamically established connections and compositions of these components.

6 Types

6.1 Designing event types

When developing with Java$_G$-TPS, the design of event types has to follow certain design guidelines, which are mostly consequences of Java's type system. More precisely, filter expression requires event types to be defined as interfaces, and similarly their method signatures to declare return types which are interfaces. Furthermore, filters with a complexity going beyond simple equality tests are hard to express.

As already mentioned briefly in the previous section, Java's `Proxy` class namely does not support class types. Therefore, all event types — and in the case of nested invocations also the types of any objects returned as results of invocations — must be interface types. This precludes the use of primitive types, and also corresponding wrapper types (e.g., `String`).

This can be circumvented in our case by introducing own classes for `String`, `Integer`, etc. with corresponding interfaces `StringIntf`, `IntegerIntf`, a workaround which is however hardly appreciated by developers.

6.2 Filter expressiveness

Furthermore, the composition of filters such as in the following example, would require operators defined on primitive types to be reflected by methods on corresponding wrapper types:

```
String company = q.getCompany();
company.equals("Telco") || company.equals("Other");
```

This sometimes comes as part of **operator overloading**, yet is not present in Java. In particular, when using dynamic proxies for content filter expression, methods returning primitive types could not be used (except `equals()`, and that only in a limited sense, since it is used as convention to express that the condition must return `true`). The following condition, though sound at compilation, could not be "registered" by a proxy, as `price` is of primitive type (and the < operator hence not reflected as a method):

```
float price = quote.getPrice();
price < 100;
```

Even when merging the two lines into one, or attempting to create an instance of `Float` from the value returned by `getPrice()` and expressing the condition with the `compareTo()` method rather than the < operator, the interception chain is interrupted.

6.3 Beyond TPS — uniformity

As shown by TPS in Java, a complex type system potentially leads to many complications when deployed at a distributed scale. While primitive types might indeed be useful in certain strongly performance-sensitive applications, and the distinction between interfaces and classes does definitely not appear to be harmful in itself, these "irregularities" lead to different semantics and hence require specific handling and implementations. They tend to represent special cases with respect to genericity and reflection, which are difficult to take into account, just like the possibility of directly manipulating fields (the main thorn in the side of creating dynamic proxies for classes in addition to interfaces). A **non-hybrid object-oriented type system** (i.e., without primitive types etc.) inherently enforcing type safety and encapsulation at compilation, i.e., avoiding direct field accesses (possibly by **automatic field access method generation**), has the advantage of more easily supporting uniform interaction, also at a distributed scale.

If features such as primitive types are really required, effort should be invested in specific support for genericity and reflection, possibly by fitting (e.g., by translation) these constructs into a more uniform underlying representation.

7 Discussion

7.1 Performance

For fairness, we analyze the effectiveness of the three presented approaches, based on the same simple architecture characterized by a *class-based* dissemination, i.e., every event class is mapped to an IP Multicast channel (more elaborate algorithms, e.g., [13], have not been tested with all TPS implementations). The test application involved three types; `Event`, its subtype `StockQuote`, and a subtype of the latter type, `StockRequest`. Since the filter evaluation is essentially the same in all three approaches, we have focused on type-based filtering.

Due to space limitiations, 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. [9] presents more detailed results.

Java-TPS and Java$_G$-TPS differ from Java$_{PS}$-TPS, in that upon publishing an event, the precise channel for the corresponding class has to be "found" first. 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 [12].

This difference is visible in Figure 7, where we compare Java$_G$-TPS (Java-TPS yielded similar results) with Java$_{PS}$-TPS. One can see that the latency of publishing an event in the case of Java$_G$-TPS is increased by runtime type checks performed to obtain the appropriate channel. The latency varies here with the number of events published subsequently (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.

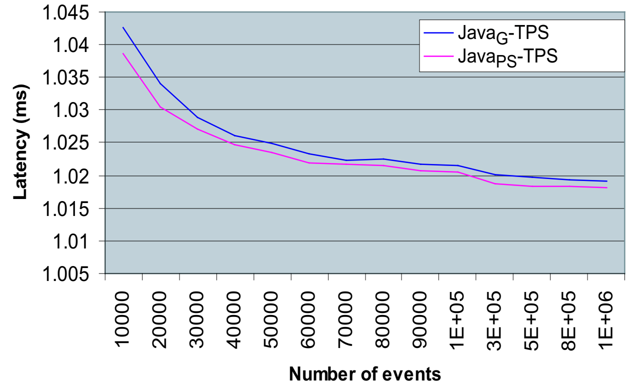

Figure 7: Java$_{PS}$-TPS vs Java$_G$-TPS

The performance of Java$_G$-TPS (and Java-TPS) is conditioned by the number of different subtypes whose instances are published through a given (G)DAC. The second set of measurements focuses on Java$_G$-TPS, and intends to compare the latencies obtained with the various event types published through a GDAC for the uppermost type. Figure 8 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. (Of course, optimizations could be performed to reduce this overhead.)

7.2 Alternative language features

In the history of programming languages, various abstractions and concepts have been proposed as first class constructs, and there are obviously no limits to what will come. We focus here on **deferred code evaluation** (see [9] for more alternatives, e.g., to genericity), as some face of behavioral reflection.

The closures used for content filter generation in Java$_{PS}$-TPS undergo a deferred **compilation**, as they are transformed at compilation into instantiations of a form

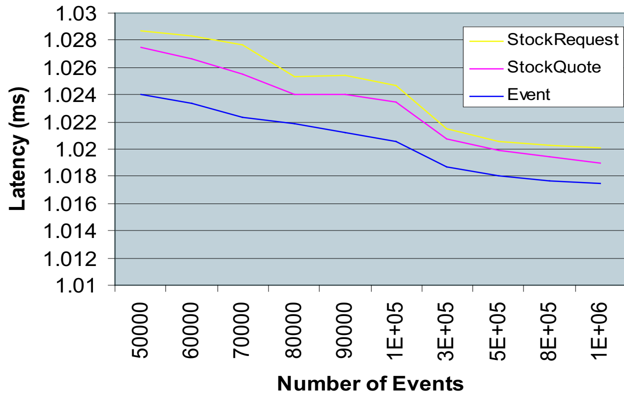

Figure 8: Different event types in Java_G-TPS

of first class abstract syntax trees. This makes these closures "migratable" and transformable, e.g., partially evaluatable. Making these closures first class would lead to an admittedly less general, but more lightweight solution to deferred code evaluation than a reification of the entire program in the form of **parse tree** such as in Smalltalk [21]. Such closures could in a more general sense be interesting for distributed abstractions. Pre-/postcoditions (*design by contract* [19]) in a distributed heterogenous RMI environment, or even queries on object-oriented databases [24], could be implemented with such a feature. Clearly, the expression of content filters reflects the need of future abstractions for seemlessly integrating with application code, and in TPS represents the most tedious part with no main-stream languages currently providing fully satisfying mechanisms.

7.3 .NET

Definitely inspired by the Java technology, Microsoft's .NET [26] 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 customizing remote invocations.

Genericity has been investigated in the context of .NET. Like in the case of Java, genericity is however not currently integrated in .NET, but rather foreseen for a future release. Similarly, .NET provides roughly the same introspection features as Java, and furthermore also provides dynamic proxies, however again restricted to interface types. However, remotely invocable objects can in .NET 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.

Hence, it appears that .NET has from the type system point of view introduced certain interesting support mechanisms. This is probably a consequence of the fact that .NET has the declared goal of providing language interoperability and hence has to be able to deal with various type systems. When considering reflection and genericity from our perspective, .NET however still plays in the same league as Java.

8 Alternative abstractions

8.1 Tuple spaces

A significant body of work sheds light on implementation strategies for tuple spaces, the spiritual ancestor of publish/subscribe which first appeared in the Linda programming language [16] as coordination means between cooperating processes.

Through a tuple space, processes can exchange arbitrary length tuples of values. Inserting a tuple into the tuple space is done using the `out` primitive. Fetching a tuple from the tuple space is done using a blocking primitive, either `in` to subsequently remove the read tuple from the tuple space, or `read` to enable the same tuple to be read by several consumers. The tuple space has since been extended with further primitives, e.g., non-blocking read primitives and callbacks. 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 space. The `var` keyword in line 4 causes the `i` to be treated as a formal parameter. The tuple added in line 1 may be extracted by line 3, and the actual value of `i` will then be 80. In line 4, the `integer` keyword simultaneously declares a variable `j` and uses it as a formal parameter.

Implementing tuple spaces in Java poses similar problems to those described for TPS. As an example, Jada [6] instruments Java with a library that supports development

of distributed applications based on tuple spaces. In Jada, class `Tuple` represents lists of Java `Objects`, and has constructors for up to 10 values. Clients thus have to cast these objects explicitly upon reception, thereby reducing type safety. 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 (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 tuple space library provides little safety, and is tedious to use.

Other approaches to tuple space interaction in object-oriented programming languages apply a different model, viewing tuples as single objects, whose fields reflect tuple attributes. A representative of that approach is given in Java by Sun's own JavaSpaces [15]. As JavaSpaces can not make use of genericity, type checks and type casts are necessary. Furthermore, encapsulation of tuples is broken by forcing fields to be declared as public and expressing and performing any content-based filtering through these fields.

Through the similarity between TPS and tuple spaces, it is easy to see that a "clean" library à la JavaSpaces could be implemented with similar features claimed for TPS.

8.2 RMI

The implementation of Java RMI can be viewed as an intermediate solution between a language-integrated RPC package and a standard 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 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 the dynamic proxies used in Section 5, to defer the binding to a remote object to runtime. This proxy mechanism seems to have been devised with the requirements of RMI in mind, as only interface types can benefit from this type of reflection. It is however not clear whether this mechanism for behavioral reflection will replace the generation of type-specific proxies through the `rmic` compiler. While a precompilation step can help dealing with language interopabilitiy, it can in the context of Java only more be motivated by performance reasons.

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 (see Section 5).

9 Concluding remarks

In the face of today's heterogenity across platforms, we believe that designers of future programming languages should foresee a more general support for component composition, and hence for distributed (and other) 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 future support. As shown in this paper by the difficulty in expressing content filters, TPS, as a paradigm emphasizing scalability and performance, requires a strong interaction with the programming language, and is hence a very demanding abstraction. Most abstractions established for distributed interaction, such as tuple spaces, or RMI, require only a subset of the features mandated by TPS.

We argue that reflection and genericity, as faces of *extensibility*, are the key concepts for a general language support of distributed programming, and that a straightforward type system can support the implementation of such features. With inherent reflective capabilities and genericity, we believe one could implement a powerful TPS library, and, as pointed out in this paper, also alternative abstractions for connecting distributed components such as tuple spaces, and RMI.

We insist on the fact that 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 for RMI, e.g., [1]). We pointed out that, from our perspective, the current support for genericity and reflection in a mainstream language such as Java is insufficient, and we illustrated how primitive types and direct field accesses contribute to these flaws. Last but not least, we also showed that .NET inherited not only approved concepts, but also weaknesses from Java with respect to reflection and genericity.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Com-

position Filters. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, pages 152–184, July 1993.

[2] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language Support for Connector Abstractions. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2003)*, pages 74–102, July 2003.

[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] A. Carzaniga, D. Rosenblum, and A. 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)*, pages 219–227, July 2000.

[6] P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 213–228. Springer, Apr. 1997.

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

[8] G. Cugola, E. D. 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)*, pages 261–270, Apr. 1998.

[9] 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.

[10] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21th International Conference on Software Engineering (ICSE '99)*, May 1999.

[11] W. Emmerich. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering - 22 nd Int. Conf. on Software Engineering (ICSE 2000)*, pages 117–129, May 2000.

[12] 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.

[13] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight Probabilistic Broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, Nov. 2003.

[14] 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.

[15] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, June 1999.

[16] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[17] C. Ghezzi. Ubiquitous, Decentralized, and Evolving Software: Challenges for Software Engineering. In *First International Conference on Graph Transformation (ICGT 2002)*, pages 1–5, Oct. 2002.

[18] 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.

[19] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, Oct. 1992.

[20] E. D. Nitto and D. Rosenblum. The Role of Style in Selecting Middleware and Underwear. In *Workshop on Engineering Distributed Objects '99, ICSE '99*, pages 78–83, May 1999.

[21] F. Rivard. Smalltalk: A Reflective Language. In *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection '96)*, pages 21–38, Apr. 1996.

[22] M. Roserens. Stock Trading with Distributed Asynchronous Collections. Master's thesis, Swiss Federal Institute of Technology, in collaboration with Lombard Odier Darier Hentsch Co., Mar. 2001.

[23] G. Steele. Growing a language. In *Addendum to the Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, Oct. 1998.

[24] D. Straube and M. Özsu. Query Optimization and Execution Plan Generation in Object-Oriented Data Management Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), Apr. 1995.

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

[26] T. Thai and H. Lam. *.NET Framework Essentials*. O'Reilly and Associates, Inc., June 2001.

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

[28] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing Middleware to Support Cross-Cutting Features in Distributed, Heterogeneous Systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, May 2003.