Distributed Programming with Typed Events

Patrick Th. Eugster

Distributed Systems and Computing Research Group
Chalmers University of Technology, Göteborg
S-412 96 Sweden
Phone: +46 702 90 30 60
Fax: +46 31 16 56 55
peugster@cs.chalmers.se

Rachid Guerraoui

Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne
Phone: +41 21 693 52 72
Fax: +41 21 693 75 70
rachid.guerraoui@epfl.ch

Abstract

Whereas the remote procedure call (RPC), including its derivates such as remote method invocation (RMI), has proven to be an adequate programming paradigm for client/server applications over LANs, type-based publish/subscribe (TPS) is an appealing candidate programming abstraction for decoupled and completely decentralized applications that run over large-scale and mobile networks. In short, TPS enforces type safety and encapsulation (just like RPC) while providing decoupling and scalability properties (unlike RPC). We present our experience gathered with two implementations of TPS in Java, namely a seminal approach relying on specific primitives added to the Java language, and a second implementation based on more general "recent" mechanisms of Java, avoiding any specific compilation.

Keywords: *Distributed programming, publish/subscribe, events, type safety, java*

1 Introduction

1.1 RPC et al.

The *remote procedure call* (RPC) [1] paradigm, including its derivates (e.g., Java RMI, CORBA, DCOM), currently represents one of the most popular paradigms for devising distributed applications. Objects (when acting as *servers*) are *invoked* remotely (by *clients*) through *proxies* (also called *stubs*). By offering the same interfaces than their respective associated remote objects, proxies hide distribution details, leading to a very convenient distributed programming style that enforces type safety and encapsulation. As widely recognized however, RPC-style interaction does not apply equally well in all contexts, since in its classic form it tends to strongly synchronize, and hence couple, the invoking and the invoked objects (Figure 1.2)[1].

1.2 Publish/Subscribe

Inspired by the *tuple space* [2] paradigm, the *publish/subscribe* [3] interaction style has appeared as a very appealing alternative to RPC in the case of asynchronous, so-called "event-based" applications, which require the dissemination of events to a potentially large number of consumers. Events are *published* by producers (*publishers*) without any knowledge of potential consumers, and are delivered to exactly those consumers (*subscribers*) who have expressed interest in (*have*

[1] Several asynchronous variants of RPC have been proposed, illustrating the severeness of this drawback.

subscribed to) those events. This paradigm offers strong scalability properties at the abstraction level, as a result of its decoupling of participants (Figure 1.2) in (1) *space* (participants do not have to be co-located nor do they require references to each other), (2) *time* (participants do not have to be up at the same time), and (3) *flow* (data reception/sending does not block participants).

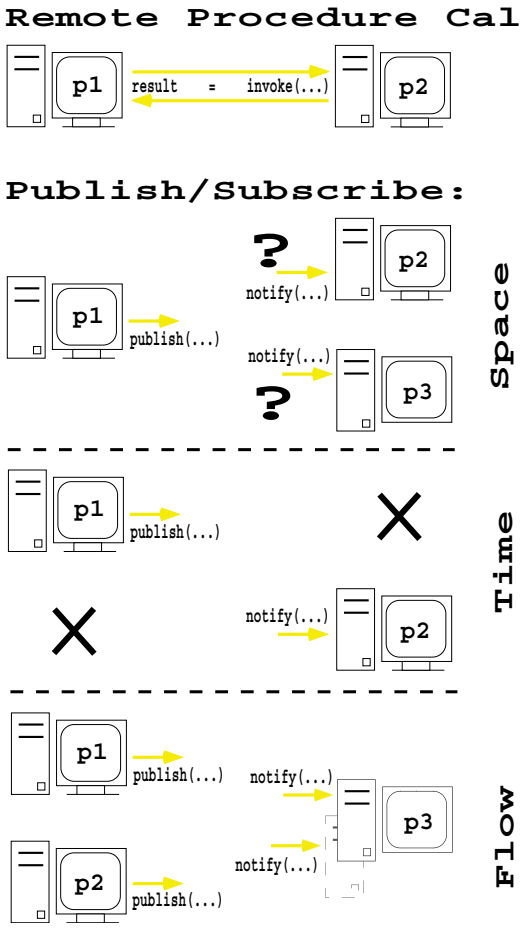

Figure 1. Coupling in RPC and publish/subscribe

1.3 The Type is the Subject

The classic publish/subscribe interaction model is based on the notion of *subjects* (or *topics*), which basically resemble the notion of groups found in distributed computing [4]. Subscribing to a subject S can in that sense be viewed as becoming member of a group S, and an event published under subject S is delivered to all members of S.

Most subject-based systems arrange subjects (e.g., "StockQuotes") in hierarchies (e.g., "/StockQuotes/TelcoMobile") and introduce wildcards to support some form of pattern matching on subject names (e.g., "/StockQuotes/*"), rendering subscriptions more expressive.

Such an addressing scheme based solely on names strongly enforces interoperability, and provides for much flexibility especially in combination with the general-purpose event types (comparable to *maps* consisting of *name-value* pairs) usually anchored as only permissible event types inside the APIs of most current systems.

With type-based publish/subscribe (TPS), events are instances of "arbitrary" application-defined event types. In essence, the TPS paradigm uses an "ordinary" type scheme without explicitly introducing a subject hierarchy, nor any other specific notion of event kind: the type *is* the subject.

Effective application events do not have to be explicitly inserted into, or extracted from, any predefined general-purpose event types, improving type safety. Similarly, consumers do not have to transform nor cast received events. General-purpose event types such as maps are merely a specific kind of events, and can still be used whenever the contents of events is not known at compilation.

1.4 The State is the Content

It is usually very convenient to adopt a *content-based* (*property-based*) publish/subscribe style, where consumers express subscriptions as *content filters* (a form of predicates) based on desired values for inherent properties of events (e.g., "value ¿ 20.2"). Most subject-based systems have been extended to support content-based filtering.

With TPS, subscriptions include content filters expressed on the public members of the types subscribed to. The content of an event object is hence implicitly defined: the state *is* the content. TPS nevertheless preserves the encapsulation of the state of event objects, by not forcing event types to reveal their state, i.e., content filters can make use of public methods. This is opposed to contemporary approaches, where applications

must define event types as sets of public fields.

In short, TPS is a high-level variant of publish/subscribe, pretty much like RPC can be viewed as a high-level variant of synchronous message passing. TPS, in contrast to the RPC however, focuses on exchanged objects rather than on the interacting objects. TPS differs from other "typed" variants of the publish/subscribe paradigm (see Sidebar 1), by preserving type safety *and* encapsulation with application-defined event types – these types being viewed as *inherent* attributes of event objects.

1.5 Example: Stock Trade

Figure 2 illustrates the intuitive idea underlying TPS, through a recurring example for publish/subscribe interaction, which is the stock trade application. A possible scenario is the following. The stock market `p1` publishes stock quotes, and receives purchase requests. These can be "spot price" requests, which have to be satisfied immediately, or "market price" requests for purchasing quotes only at the end of the day, or once another given criterion is fulfilled. As outlined in Figure 2, these different kinds of events result in corresponding event types, rooted at the `StockEvent` type (details of the elaborate event types are omitted here for simplicity). These event types are part of the application design.

Market price requests can however expire, and for the broker's (e.g., `p2`) convenience, an intermediate party (`p3`), e.g., a bank, might also handle such requests on behalf of her/him, for instance by issuing spot price requests to the stock market once the broker's criteria are satisfied. Figure 2 illustrates this through `p2`, which expresses only interest in stock quotes that cost less than $100.

Note that by subscribing to a type `StockEvent`, `p3` receives instances of its subtypes `StockQuote` and `StockRequest`, and hence all objects of type `SpotPrice` and `MarketPrice`.

1.6 Roadmap

The next two sections are devoted to presenting TPS from an abstract viewpoint. The two

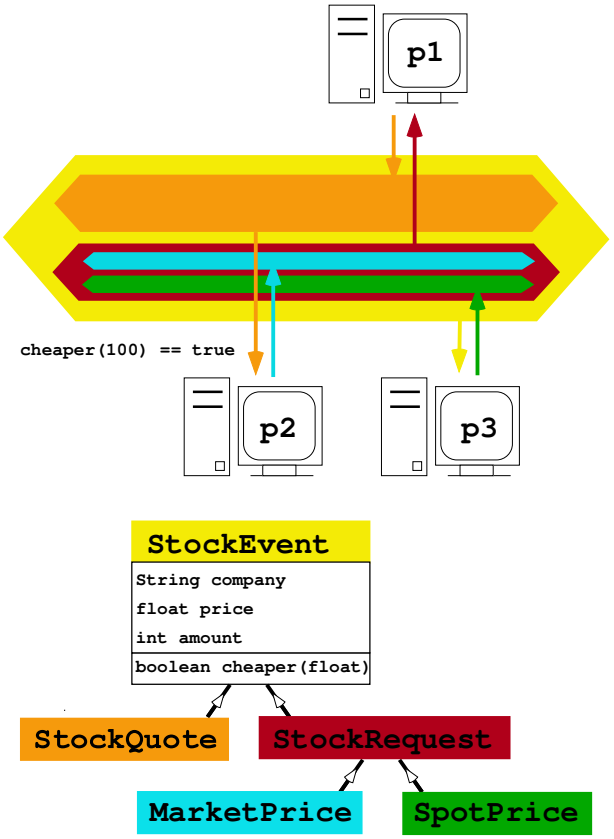

Figure 2. Type-based publish/subscribe

following sections each report on an implementation of TPS in Java. For presentation simplicity, and to better illustrate TPS, we start by presenting a very concise implementation of TPS relying on an extension of Java with specific primitives for TPS programming. We then present an implementation of TPS in a "recent" variant of the Java language, thereby ensuring type safety just like encapsulation without however requiring instrumented compilation. This article concludes with a note on interoperability.

2 Publishing Event Objects

The only contracts between publishers and subscribers are the types of the published events. A publisher has no explicit notion of "destination" when publishing an event. The set of destinations is implicitly and dynamically defined by the subscribers whose criteria match that event object.

2.1 Distributed Object Cloning

With a published event `e` acting as template, a publication can be pictured as a distributed form of object cloning, where a clone of the prototypical object `e` is created for every subscriber. The set of processes where this action will take place is given by the set of processes who are willing to host such objects, i.e., who's subscription criteria match the template object. Inversely, a subscription expresses the desire of getting a clone of every published object which corresponds to the subscription criteria.

The notion of cloning here corresponds to a *deep* cloning: when a clone of an object is created, its fields are recursively cloned.

2.2 Sending Objects Over the Wire

This deep cloning is implicitly given by the *serialization* that is applied. Published event objects are *serialized*, i.e., they are traversed, and their state extracted and used to generate a representation more suitable for the underlying communication layers, which take care of transmitting these as "messages" to every process hosting matching subscribers. There, these messages are *deserialized* to instantiate new objects.

3 Subscribing to Event Types

The main subscription criterion for consumers is the (abstract) type of the event objects of interest. When subscribing to a type T, one expresses interest in instances of T, that is, instances of any types which *conform to* T.

3.1 Event Type System

The interpretation of *conformance* depends naturally on the considered type system. A type system for events can be derived from a single programming language, leading to a first class TPS package comparable to a first class RPC package like Java RMI in the case of the Java programming language. An event type system can as well be based on a neutral *event definition language* (EDL) to enforce interoperability, leading to a second class TPS [6].

In any case, object types offer richer semantics than just information about inclusion relationships. An object type encompasses contracts guiding the interaction with its instances: an interface composed of public members describing its incarnations.

3.2 Types for Fine-Grained Subscriptions

This information can be naturally used to express more fine-grained subscriptions, that is, encompassing content filters. Ideally, when expressing content filters, the *full* semantics of the programming language in which they are expressed can be exploited. Consider the stock market example introduced before-hand. Stock quotes are published by the stock market, and are received by brokers. Stock quote events carry a set of fields, like the amount of stock quotes and their price. (Figure 3 shows the Java class `StockQuote` corresponding to simple stock quotes.)

A subscription expressing interest in all stock quotes of Telco Mobiles with a price less than $100 could be expressed like the following, supposing that `q` is a formal argument representing an instance of type `StockQuote`:

```
q.cheaper(100) &&
q.getCompany().equals("Telco Mobiles")
```

Expressing filters in such a convenient way considerably relieves the burden on programmers, by avoiding the learning of a specific subscription language (à la SQL), and by enforcing static type checking and hence decreasing the risk of runtime errors. Furthermore, encapsulation of event objects is preserved, as methods are used to describe content filters, which is not the case in related approaches (see Sidebar 1). One could further make use of exceptions (`try...catch` clauses around the above lines), arbitrary language statements (`if`, `while`, etc.), parameters by passed by reference rather than by value, etc.

3.3 Issues with Content Filters

Certain restrictions on the semantics of content filters are however inevitable to ensure an efficient and scalable implementation of the underlying TPS engine. As widely recognized, the more expressive filters are, the more it becomes difficult to analyze and optimize these filters [5], which in turn makes effective distributed filtering and routing of events hard. Filtering is usually performed by a distributed overlay network formed by hosts acting as application-level routers. Content filters might hence have to be transferred to foreign hosts, where they can be regrouped and redundancies can be factored out [7]. This requires an insight into these filters, which becomes increasingly difficult as the programming language becomes involved, and becomes even harder when interoperability is emphasized (see Section 6).

The following two sections illustrate two ways of dealing with these issues in Java; (1) by extending the Java programming language (and compiler) for the sole purpose of inherently supporting TPS, and (2) by relying on a library implemented with a "recent" version of Java with support for *genericity*[8].[2] The expression of the above content filter example in the respective approaches is illustrated in Sidebar 2.

4 A Language Integration Approach to TPS

Java$_{PS}$ [9] is an extension of Java, integrating TPS by adding two specific primitives `publish` and `subscribe` to the Java language.

4.1 Publishing

An event object `e` is published by making use of the `publish` primitive, leading to the simple syntax:

```
publish e;
```

This statement triggers the creation of a copy of `e` for every subscriber.

[2] Genericity is foreseen for Java 1.5. We made use of the compiler underlying Sun's current efforts.

```
/* stock quote events */
import java.io.Serializable;

public class StockEvent implements Serializable {
  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 boolean cheaper(float thanPrice)
    { return (price < thanPrice); }
  public StockEvent(String company, float price,
                    int amount)
  {
    this.company = company;
    this.price = price;
    this.amount = amount;
  }
}

public class StockQuote extends StockEvent {
  private long time;
  public long getTime();
  public StockEvent(String company, float price,
                    int amount, long time)
  {
    super(company, price, amount);
    this.time = time;
  }
}
```

Figure 3. Stock events

4.2 Subscribing

A subscription to a type `T` takes roughly the following form:

```
Subscription s = subscribe (T t) {...} {...};
```

The first expression enclosed in brackets is a block, provided by the application, which represents a content filter for events of the subscribed type `T` (expressed through a formal argument called `t` here). A boolean value is returned, indicating whether the event is of interest or not. The second block represents an event handler which is evaluated every time an event successively passes the filtering phase. The same formal argument `t` represents the event of interest in this case. A subscription handle is returned by a subscription

expression. It allows, among other things, the setting of *Qualities of Service* (QoS) parameters, or the activation and deactivation of a subscription:

```
s.activate();
...
s.deactivate();
```

Content filters are hence expressed through the programming language itself, yet to enforce the mobility of these filters, these filters can only support a subset of the semantics of Java. Java$_{PS}$ relies on a specific compiler, whose main added functionalities are required to generate abstract syntax trees from these content filters.

5 A Library Approach to TPS

Generic Distributed Asynchronous Collections (GDACs) [10] constitute a library approach to TPS, meaning that GDACs are implemented without any TPS-specific compiler.

5.1 GDACs

Just like any collection, a GDAC is an abstraction of a container object that represents a group of objects (Figure 4, the `GDAC` type extends the standard Java `Collection` type). A GDAC can however also be viewed as an abstraction of an event channel, where elements are events.

"G" for "Generic": To enforce type safety, a GDAC represents a specific type of events. Strong typing is therefore possible and is enforced to avoid explicit type casts, and hence potential runtime errors. In other terms, the interface offered by a GDAC for a given Java type `T` offers methods where parameters representing event objects are of type `T`. Generating a typed GDAC for every event type can be avoided by using *genericity*, which allows to represent the event type handled by GDACs as a *type parameter* (Figure 4).

"D" for "Distributed": Unlike a conventional collection, a GDAC is a distributed collection whose operations might be invoked from various nodes of a network, in a way similar to a shared memory. GDACs are not centralized entities with remote access capabilities, but are *essentially* distributed to guarantee their availability despite certain failures. Participants act with a GDAC through a local proxy, which is viewed as a local collection and hides the distribution of the GDAC.

"A" for "Asynchronous": GDACs promote an asynchronous interaction style. When *adding* an element to a GDAC (`add()` method in Figure 4), the call can return before the element has appeared on all involved nodes. By querying a GDAC for the presence of new elements (overloaded `contains()` method from standard Java collections), one expresses an interest in *future elements*. A client expresses its interest in such future objects by registering a *callback* object with the GDAC, through which the client will be notified of objects "pushed" into the GDAC.

5.2 Generic Java

While languages like C++ or Ada 95 incorporate *generic* types [11], languages like Java or Oberon have been initially designed to replace variable types by the root of the type hierarchy. For such languages lacking generic types and methods, including Java, adequate extensions have been widely studied.

We implemented the first GDACs for TPS based on *Generic Java* (GJ) [8], the most prominent among a multitude of dialects of the Java language with genericity. As a strict superset of Java, GJ comes with a specific compiler, which is fully compatible with the Sun release, and enables the use of the original Java virtual machine.

6 A Note on Interoperability

The experiences presented so far have all been conducted with the Java programming language, and it has turned out that providing interoperability for TPS involves more delicate issues than in the case of RPC. Though, just like TPS, RPC relies on the invocation semantics and type systems

```
import java.io.Serializable;
import java.util.Collection;

public interface GDAC<T extends Serializable>
    extends Collection<T> {
  /* inserting an element: publishing */
  public boolean add(T t);
  /* query the local collection proxy */
  public boolean contains(T t);
  ...
  /* query the global collection: subscribing */
  public Subscription<T> contains(Subscriber<T> s);
  ...
}

public interface Subscriber<T> {
  public void notify(T t);
}

public class Subscription<T> {
  public T getProxy();
  ...
  public void activate()
    throws CannotSubscribeException;
  public void deactivate()
    throws CannotUnSubscribeException;
}
```

Figure 4. Interfaces related to GDACs

of the supported programming languages, it seals (in most cases) distinct address spaces from each other, letting only invocations enter and exit. TPS on the other hand, does not occasion the *invocation* of coarse-grained remote objects, but rather relies on the *transfer* of fine-grained remote objects, which might require the transfer of the code of such transferred objects. Similarly, applying content filters remotely leads to the necessity of migrating and "interpreting" code. Interoperability in the case of TPS hence requires further assumptions, like a common intermediate programming language (e.g., byte-code [6]), or that all event types are implemented in all involved languages.

References

[1] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[3] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principles*, pages 58–68, December 1993.

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

[5] 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), pages 219–227, July 2000.*

[6] 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), to appear May 2003*

[7] 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 1999), pages 53–62, November 1999.*

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

[9] P.Th. Eugster, R. Guerraoui, and Ch. Heide Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-*

Oriented Systems, Languages, and Applications (OOPSLA 2001), October 2001.

[10] 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)*, pages 252–276, June 2000.

[11] R. Milner. A theory of type polymorphism in programming. *Journal of Computing Systems Sciences*, 17:348–375, 1977.

Sidebar 1 On Types and Events

We outline some of the most prominent approaches to distributed programming with some form of typed events.

COM+

Microsoft's COM+ [1] promotes a model based on the types of *subscribers* rather than on the types of events: similar to RPC, objects can provide specific interfaces defining own methods through which they will be invoked. Applications must provide typed dummy proxies that publishers invoke. At runtime, these invocations are then intercepted by the event service and forwarded to those subscribers implementing the same type as the proxy. To respect the asynchronous nature of event-based programming based on the publish/subscribe paradigm, such methods are not allowed to return results.

Method invocations hence play the role of events, the "content" of these events being made up of the actual arguments. Content filters in COM+ are obtained by specifying admissible values for invocation arguments of methods, and are expressed through a limited subscription grammar.

CORBA Event Service

The OMG has specified a CORBA service for publish/subscribe-oriented communication, known as the CORBA Event Service [2]. According to the general service specified, a consumer registers with an *event channel* expressing thereby an interest in receiving *all* the events from the channel. These channels are named objects, coming close to non-hierarchical subject names.

A form of typed interaction is provided, similar to the model in COM+, enabling the use of the types of consumers, but also of producers (the CORBA Event Service supports pull- and push-style interaction), as main subscription criterion. According to the type of interaction, methods only have input parameters or return values to respect the asynchronous nature of publish/subscribe. Typed proxies are generated based

on the application's interface, which in practice requires a specific compiler.

TAO Event Service

Shortly after commercial implementations of the CORBA Event Service became available, several deficiencies (e.g., missing support for QoS and realtime requirements, difficulties with the abovementioned typed events) became apparent, leading to extended event service implementations, one of the most significant being the one used in the TAO Realtime ORB [3]. It addresses mainly realtime issues, but also enforces subscriptions based on the identity of the publisher and/or the event types. In the latter case, the "type" is an integer value explicity assigned to every event by storing it in a particular field.

CORBA Notification Service

Based on the observed lacks of the CORBA Event Service, the OMG issued a request for proposal for an augmented specification, the CORBA Notification Service [4]. A *notification channel* is an event channel with additional functionalities, including notions of priority and reliability. A new form of semi-typed events, called *structured events* is introduced. These represent general-purpose event types, which manifest fields like event type and event name, and are roughly composed of an event header and an event body. Both parts consist each of a fixed part and a variable part.

The variable parts of structured events (as well as the fixed header part) are composed of name-value pairs, for which the specification mentions a set of standardized and domain-specific compositions. Standard properties include a notion of event *type*, however represented by a name.

In the context of content filtering, these name-value pairs are used to describe content filters, called *filter objects*. These are described as strings following a complex subscription grammar called the Default Filter Constraint Language, which are interpreted at runtime and hence offer little safety.

Java Message Service

The Java Message Service (JMS) [5] is Sun's answer to the CORBA Event & Notification Service Specifications. Different types of events are predefined, varying by the format of their body, yet all inheriting from a basic event type representing a map for name-value pairs. A set of keys are predefined, including a property representing the event type, however, just like in the case of the CORBA Notification Service specification, consisting simply of a type name. Content filters are called *message selectors*, and are once more expressed through strings based on a SQL-like grammar.

JavaSpaces

The JavaSpace [6], Java's variant of the tuple space originally introduced in Linda, is a container of objects that can be shared among various producers and consumers. When consumers register callback objects with a JavaSpace, one ends up with a publish/subscribe communication scheme in which the JavaSpace plays the role of the event channel aimed at multicasting event notifications to a set of subscriber objects. Custom event types subtype the `Event` type, adding publicly accessible fields. A given subscriber of a JavaSpace advertises the type of events it is interested in by providing a template object `t`. A necessary condition for `o`, an object notifying an event, to be delivered to that subscriber, is that `o` conforms to the type of `t`. Furthermore, the field values of `t` have to match the corresponding field values of `o`, with `null` playing the role of wildcard.

ECO

An approach to integrating event-based interaction with C++ is discussed in the ECO (events + constraints + objects) model [8]. Events are added as specific language constructs decoupled from the main application objects, necessitating a considerable number of language add-ons. Filtering can be based on the publisher's identity (the source), and several types of *constraints*. *Notify constraints* are expressed based on the fields of

events, and *pre-* and *post constraints* use the state of the subscriber object. Methods can however not be used to express constraints.

CEA

The Cambridge Event Architecture (CEA) [7] is based on an interoperable object model, in which event types are described through the ODMG's Object Definition Language (ODL). C++ and Java mappings for this language are mentioned. Precompilers generate specific adapters (called stubs in the CEA) for exchanging typed events. Filtering mechanisms are also included, however once more based on viewing the events as sets of fields.

References

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

[2] OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service.* OMG, March 2001.

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

[4] OMG. *Notification Service Standalone Document.* OMG, June 2000.

[5] Sun. *Java Message Service Specification.* Sun, February 2002.

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

[7] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel and M. Spiteri. Generic Support for Distributed Applications. IEEE Computer 33(3):68–76, March 2000.

[8] 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 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 83–92, June 2000.

Sidebar 2 TPS Programming

We illustrate programming with TPS first through Java$_{PS}$, a variant of Java including primitives for TPS, and second, through GDACs, a library for TPS based on Java with the recent compiler support for genericity.

Programming with TPS Language Primitives

Programming with TPS language primitives can best be illustrated through the stock market scenario introduced previously. The following example reuses the `StockQuote` event class of Figure 3.

The stock market can publish a stock quote event by doing something like the following:

```
StockQuote q =
  new StockQuote("Telco Mobiles", 80, 10, 1500);
publish q;
```

Below, we give an example of a subscription, which expresses an interest in all stock quotes of Telco Mobiles with a price less than $100.

```
Subscription s = subscribe (StockQuote q)
  {
    return (q.cheaper(100) &&
      q.getCompany().equals("Telco Mobiles"));
  }
  {
    System.out.print("Got offer: ");
    System.out.println(q.getPrice());
  };
s.activate();
```

Programming with a TPS Library

With GDACs, stock quotes can simply be published by adding them to a GDAC parameterized by the stock quote type:

```
GDAC<StockQuote> qs = new GDASet<StockQuotes>();
StockQuote q =
  new StockQuote("Telco Mobiles", 80, 10, 1500);
qs.add(q);
```

Expressing a subscription requires slightly more effort. A subscriber type has to be created explicitly to handle events:

```
public class StockQuoteSubscriber
  implements Subscriber<StockQuote> {
    public void notify(StockQuote q) {
      System.out.print("Got offer: ");
      System.out.println(q.getPrice());
    }
}
```

Filters are expressed through the `Subscription` class which has richer semantics than its homonym used in the language integration approach. The `contains()` method used for subscribing returns an instance of that class, through which a content filter can be expressed by making use of *proxies* introduced for *behavioral reflection* [1] in Java 1.3. Such a proxy enables to "record" the invocations performed on it; it can be used as a form of formal argument, which enables the expression of *what* invocations are to be performed on a filtered event. The following example illustrates the use:

```
GDAC<StockQuote> qs = new GDASet<StockQuote>();
Subscription<StockQuote> s =
  qs.contains(new StockQuoteSubscriber());

StockQuote q = s.getProxy();
q.cheaper(100);
q.getCompany().equals("Telco Mobiles");
s.activate();
```

Here, we express the same subscription than in the language integration approach. By convention, a logical *and* of two constraints is indicated by expressing both constraints through the same proxy, as above, while a logical *or* requires the creation of two proxies, one for each constraint. More expressive content filters require additional functionalities in the `Subscription` class. These lead to additional complexity and reduced safety, which are mainly a consequence of the fact that Java is a *hybrid* object-oriented language, i.e., it provides primitive types. Moreover, Java only supports proxies for abstract types, i.e., interfaces. To be able to implement the presented examples, we had to extend the Java proxy implementation to support proxies for classes.

References

[1] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.*

MIT Press, 1991.

Biographies

Patrick Th. Eugster is a postdoctoral researcher at Chalmers University of Technology in Göteborg, Sweden. His interests include algorithms and abstractions for distributed programming. He received an MS in computer science and a PhD in communication systems, both from the Swiss Federal Institute of Technology in Lausanne.

Rachid Guerraoui is professor in computer science at the Swiss Federal Institute of Technology in Lausanne. He is interested in distributed and object-oriented programming. He received an MS and a PhD in computer science from the universities of Jussieu and Orsay, respectively.