Linguistic Support for Large-Scale Distributed Programming

Patrick Th. Eugster   Rachid Guerraoui   Christian H. Damm

Distributed Programming Laboratory
Swiss Federal Institute of Technology, Lausanne
CH-1015 Lausanne, Switzerland

Department of Computer Science
University of Aarhus
8200 Aarhus N, Denmark

Abstract

This paper presents linguistic primitives for publish/subscribe programming using events and objects. We integrate our primitives into a strongly typed object-oriented language through four mechanisms: (1) serialization, (2) multiple subtyping, (3) closures, and (4) deferred code evaluation. We illustrate our primitives through Java, showing how we have overcome its respective lacks. A precompiler transforms statements based on our publish/subscribe primitives into calls to specifically generated typed adapters, which resemble the typed stubs and skeletons generated by the rmic precompiler for remote method invocations in Java.

1 Introduction

1.1 RPC et al.

One of the most popular styles of distributed programming relies on extending the notion of invocation to a distributed context, i.e., offering some form of *remote procedure call* (RPC) [BN84]. The integration of this distributed interaction style with an object-oriented programming language has been thoroughly studied, e.g., [Lis88, CDJ$^+$89, Lis93, Car95]. More recently, Java [GJS00] has introduced its own variant, the *remote method invocation* (RMI) [Sun99b], through a precompiler approach.

By using the same abstraction for distributed interactions as for local ones, RPC and its derivatives integrate naturally with a language, and make distributed programming look simple.

1.2 Publish/Subscribe

Motivated by the observation that RPC is not always the best solution, the integration of alternative distributed interaction styles, like *asynchronous* RPC (e.g., [YBS86, LS88]) or the *tuple space* [Gel85] paradigm (e.g., [MK88]) with a language have also been studied. More recently, Oki et al.[OPSS93] have pointed out the existence of many distributed "event-based" applications for which a *publish/subscribe* interaction style is very appealing, thanks to its strong decoupling of participants in (1) *time* (participants don't have to be up at the same time), (2) *space* (participants don't have to know each other), and (3) *flow* (data reception/sending does not block participants).

Several authors suggested libraries for distributed interaction based on publish/subscribe (e.g., JavaSpaces [FHA99], SmartSockets [Cor99], Distributed Asynchronous Collections [EGS00]), also implementing standardized API's (CORBA Event & Notification Services [OMG01b, OMG00], Java Message Service [HBS98], etc.), but to our knowledge, very little effort has been done to inherently support publish/subscribe in a language.

1.3 Linguistic Support for Publish/Subscribe

The goal of this paper is precisely to explore the ramifications of integrating publish/subscribe primitives into a strongly typed object-oriented language.[1]

Our linguistic primitives have been designed based on four simple principles:[2]

LP1 *Type safety.* In a strongly typed language, strong typing is enforced in local interactions, and as far as possible, should also be enforced for remote object interaction. Type errors should be recognized

[1] The goal of this paper is not to advocate for a library vs an integration approach in the case of publish/subscribe, but rather to explore the ramifications of an integration approach (see Section 7).

[2] These principles have resulted from our previous experiences around objects and publish/subscribe (e.g., [EGS00, EG01]), including applications we have helped devising in the domains of banking and telecommunications. Obviously, these principles are not exclusive; other application domains might identify different requirements.

at compilation, alleviating the already cumbersome debugging of distributed applications.

LP2 *Encapsulation preservation.* Events are to be considered as objects, and hence as instances of abstract types. Their implementation details should not be revealed, and should not be systematically used to describe subscription criteria.

LP3 *Application-defined events.* Events should be definable by the application, with minimal imposed design choices.

LP4 *Composable event semantics.* To express some form of *Qualities of Service* (QoS), different semantics should be assignable to events, and these semantics should be composable.

Combining these principles is not straightforward. For instance, combining (1) *content-based* subscription based on event properties and (2) preservation of encapsulation of event objects (i.e., not systematically expressing subscriptions as *attribute-value* pairs, ruling out any query languages such as SQL) is already commonly pictured as a contradiction per se [Koe99]. Further complexity is added by requiring (3) transparency of subscriptions, i.e., giving the underlying publish/subscribe system full access to subscription criteria in order to optimize the filtering of events (e.g, by factoring out redundancies between subscriptions of different subscribers [ASS+99]).

1.4 Contributions

We present two language primitives for the expression of *type-based* publish/subscribe programming: `publish` and `subscribe`.

Instead of introducing a new programming language, we illustrate our three primitives through the well-known general-purpose Java language. Along the way, we identify a set of four mechanisms, which provided by a language, strongly enforce its support for publish/subscribe; roughly *serialization*, *multiple subtyping*, *closures*, and *deferred code evaluation.* These are however not to be viewed as *sufficient* nor as *necessary* conditions, as shown by Java, which does not incorporate all these mechanisms.

Our solution for the Java language relies on a precompiler, which transforms our specific constructs to invocations on specifically generated typed adapters.

In that sense, our precompiler can be seen as the publish/subscribe counterpart to the Java RMI compiler, and we show that the two interaction paradigms are in fact not contradictory, but that a combination of these two paradigms can be seen as a powerful tool for devising distributed applications.

1.5 Status

Our primitives were implemented using the infrastructure offered by the *Distributed Asynchronous Computing Environment* (DACE). This infrastructure has already been the base for a library approach to integrating publish/subscribe with Java [EGS00, EG01] along similar principles, making however use of other mechanisms to their achievement, namely *reflection* and *parametric polymorphism* together with *serialization.*

The goal of this paper is to emphasize language issues, leaving aside issues related to distribution; the implementation of our distributed architecture, its underlying algorithms and their performance and scalability have been addressed in other publications, e.g., [EGH+01, EBG01].

Supported by our two complementary approaches, we believe that the current work is not intrinsically tied to our architecture, but could be deployed on most existing publish/subscribe systems.

1.6 Roadmap

This paper is structured as follows: Section 2 introduces the type-based publish/subscribe variant which has allowed us to combine objects and events. Section 3 discusses the semantics of our primitives and their syntax in Java. Section 4 presents the implementation of our primitives. Section 5 discusses several issues, like design alternatives. Section 6 discusses related work. Section 7 concludes this paper.

2 Type-Based Publish/Subscribe

This section introduces our model of events and objects, and the *type-based* variant of publish/subscribe intrinsically coupled with that model. We depict the effects of subscribing to types in Java, and give a rough idea of our primitives for publish/subscribe interaction along with a simple example.

2.1 Model

The core idea underlying our integration of publish/subscribe with objects consists in viewing events as first class citizens, and subscribing to these events by explicitly specifying their type.

2.1.1 Obvents

By considering events as first class citizens, that is, not as specific constructs (e.g., [HMN+00]), but as specific application-defined objects, we strongly enforce the support of *LP3*. To emphasize the object nature of events, we call these *event objects*, or to abbreviate notation, simply *obvents*.

Similarly to [OPSS93], we distinguish mainly between two categories of objects, but introduce two further (sub)categories.

Unbound objects: Unbound objects are *locality-unbound*, that is, their semantics do not depend on any local resource. Such objects could be serialized and transferred to another address space (in [OPSS93] these are termed *data objects*).

Obvents: Obvents represent a specific kind of unbound objects. Such objects are used to notify events, and can in a nested way, contain other unbound objects

Bound objects: These objects are *locality-bound*, i.e., they are tied to an address space remain in that address space during their entire lifetime. They may make use of local resources (*service objects* in [OPSS93]).

Notifiables: Potentially, any bound object could take the role of subscriber, but in general, only particular objects subscribe to obvents.

An obvent class can best be pictured as a factory for instances incarnating notifications for events of the same kind, i.e., from the same *event source*. A notification from such an event source is reified through an obvent; basically an object which is serialized and sent over the wire to a set of destinations, where each copy is deserialized.

Note that we do not introduce a specific publisher type. Any object (bound but also unbound) can publish obvents. The notifiables introduced to represent subscribers will mainly be used for illustration purposes. As we will see shortly, the application will not be explicitly dealing with such objects, but will view these as *obvent handlers*, or simply *handlers*.

2.1.2 Effects of Publishing Obvents

Publishing an obvent `o` can thus be understood as some form of distributed object creation, where the created objects are clones of `o` which acts as template. More precisely, a distinct copy of a published obvent is created for *each* subscriber:

Obvent Global Uniqueness: Suppose an obvent `o1` published from an address space `a1`: if an address space `a2` contains two notifiables `n1` and `n2`, these will receive references to two new distinct clones of `o1`, say `o2` and `o3`.

Obvent Local Uniqueness: In the above scenario, if the address space `a1` also contains a notifiable `n3`, then `n3` will receive a reference to a new obvent `o4`.

A subscription can in that sense be seen as a contract for hosting objects created as copies of published objects. Note that if the same obvent is published twice, two distinct copies will be created again for every subscriber.

2.1.3 Type-Based Subscription

By using the type of obvents as *basic* subscription criterion, we strongly enforce the integration of publish/subscribe interaction into a language in a way that respects type safety (*LP1*): by matching the notion of *event kind* with that of an *event type*, i.e., using the type scheme of the programming language as subscription scheme, the type of the received events is known, and compile-time type checks can be performed.

Figure 1 illustrates the intuitive idea underlying our approach, through a recurring example for publish/subscribe interaction, which is the stock trade application. A possible scenario is the following. The stock market, here denoted by `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. Latter requests can however expire, and for the broker's (such as `p2`) convenience, an intermediate party (`p3`), e.g., a bank, might also handle such requests in behalf of her/him, for instance by issuing spot price requests to the stock market once the broker's criterion is satisfied.

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

2.2 Type-Based Publish/Subscribe in Java

In many strongly typed object-oriented languages like C++ [ES92] or Eiffel [Mey92], the inheritance hierarchy determines the conformance (subtype) relation. In such type schemes, the notions of *type* (*abstract type*, *type definition*, *interface*, *signature*) and *class* (*concrete type*, *type implementation)* are identical.

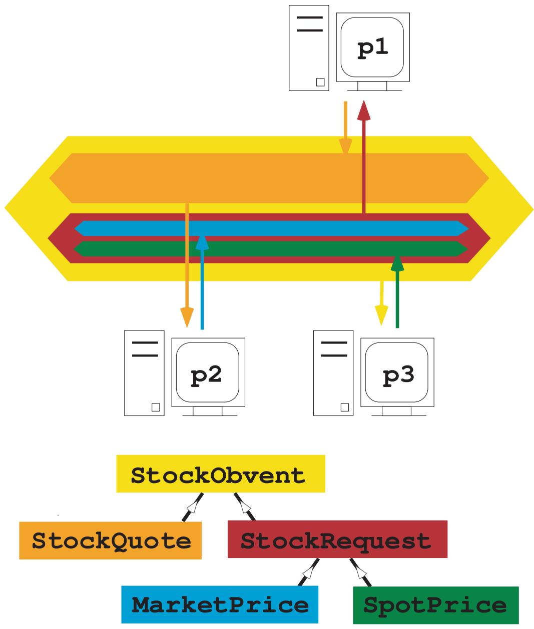

Figure 1. Type-based publish/subscribe

The Java type system is inspired by the separation of inheritance and subtyping in the sense of [CHC90]. To avoid problems known from multiple inheritance, Java offers only simple inheritance, yet introduces multiple subtyping through interfaces. In Java, types can be defined in the following two ways:

Explicit declaration: A type can be explicitly declared by declaring an interface, which can subtype several superinterfaces: an interface `I1` which extends another interface `I2` represents a subtype of the type declared by `I2`.

Implicit declaration: Defining a class `C` implicitly declares a type, and at the same time gives the class which implements it. If a class `C1` inherits from another class `C2`, then the type defined by `C1` is a subtype of the type of `C2`. A class can subtype multiple interfaces: for any interface `I` implemented by a class `C`, the type defined by `C` is a subtype of `I`.

Note that a class `C` which implements a single interface `I` without adding any new methods also defines a *new* type, which is a subtype of `I`'s type.

As a consequence of the intertwining of types and classes in Java, it must be possible to subscribe to interfaces as well as to classes.

2.3 Expressing Type-Based Publish/Subscribe

To express event-based distributed interaction based on the type-based publish/subscribe paradigm, we introduce two primitives. We give here an abstract overview of these, and give more details in Section 3.

2.3.1 Publishing

An obvent `o` is published through a primitive `publish`, leading to the simple syntax:

```
publish o;
```

This statement triggers the creation of a copy of `o` for every subscribed object, according to the rules described beforehand. In that sense, the `publish` primitive can be seen as a distributed variant of the `new` primitive found in many languages. Syntactically, the `publish` primitive bears more resemblances with the very common `return` primitive, or the `throw` primitive for the raising of an exception in Java.

2.3.2 Subscribing

In attempt to satisfy *LP1*, *LP2*, and *LP4*, we chose the obvent type as the *basic* subscription criterion. A pure static subscription scheme, like topics (e.g., [OPSS93, Ske98, AEM99, Cor99, TIB99], also called *subjects*) or types in our case, has been shown to offer only limited expressiveness. This observation has motivated *content-based* publish/subscribe (e.g., [ASS+99, CNF98, Car98, SA97], also called *property-based* publish/subscribe), where a subscription takes properties of obvents into consideration. When subscribing, the desired properties are expressed through a predicate, or *filter*. We thus combine a subscription to a type `T` with the declaration of such a filter:

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

The first expression enclosed in brackets represents a block, provided by the application, which expresses how to handle obvents of type `T` (represented by a formal argument called `t` here) in order to return a boolean value indicating whether the obvent is of interest or not. The second expression is a block which is evaluated every time an event successively passes the filtering phase, and represents thus the notifiable object. The same formal argument `t` represents the event of interest in this case. A subscription handle is returned by a subscription expression. It gives the possibility to identify a subscription, activate and deactivate it.

The motivation of capturing the code for filtering in a closure is to delay its evaluation: to avoid redundant filtering, as well as wasting network bandwidth, it is interesting to apply filters on foreign hosts, which are possibly entirely dedicated to filtering. By gathering filters of several subscribers on a given host, a compound filter can be generated which factors out redundancies between these individual filters. By doing so, performance can be significantly improved (e.g., [ASS+99]).

The use of a closure also to capture the code applied for the evaluation of received events enables the avoiding of a callback mechanism, which greatly enforces type safety and has the great advantage of regrouping code related to a subscription in a single succinct expression.

Note that one can easily subscribe to *all* obvents of a type `T` by doing something like the following:

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

2.3.3 Example

Consider the stock market example introduced above. Stock quotes are published by the stock market, and are received by brokers. Stock quotes carry a set of attributes, like the amount and price of the stock quotes. Figure 2 shows the Java code for simple stock quotes and stock quote subscribers.

```
/* stock quote obvents */

public class StockObvent implements Obvent {
  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 StockObvent(String company, float price,
                     int amount)
  {
    this.company = company;
    this.price = price;
    this.amount = amount;
  }
}

public class StockQuote extends StockObvent {
  public StockObvent(String company, float price,
                     int amount)
  { super(company, price, amount); }
}
```

Figure 2. Stock quote notifications

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

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

Below, we give an example of a subscription, which expresses an interest in all stock quotes of the *Telco* group with a price less than 100$:

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

It can easily be seen that the stock quote published in the above example satisfies these criteria.

3 Java$_{ps}$

This section illustrates mechanisms which strongly support the implementation of publish/subscribe in a language according to our model. We discuss the syntax and precise semantics of our language primitives, and informally show how these would fit into the Java language (leading to a new instance of *StatementWithoutTrailingSubstatement*, and a new *PrimaryNoNewArray* expression, § 14.5 and § 15.8 in [GJS00] respectively). We refer to this extension as Java$_{ps}$. The classes and interfaces related to our approach are regrouped in a package `java.pubsub`.

3.1 Inside Obvents

Obvents are objects that are serialized, sent over the wire, and deserialized. Java incorporates a default serialization mechanism, which can be exploited by subtyping `java.io.Serializable`.

3.1.1 Basic Type

The basic Java `Obvent` type (Figure 3) thus subtypes that type. This eases the implementation of our obvent model in general, and we state this as a first mechanism which enforces the realization of our model in a language:

LM1 *Default serialization mechanism.* A language-provided serialization/deserialization mechanism eases the transformation of event objects into conveyable low-level messages.

This principle strongly supports *LP3*: with a default serialization mechanism, developers can be relieved from the burden of implementing specific operations or hooks in their obvents. The design phase of obvents can be cut down to the essential meaning of the event.

```
package java.pubsub;

import java.io.*;

/* obvents */

public interface Obvent extends Serializable {...}
public interface Reliable extends Obvent {}
public interface Certified extends Reliable {}
public interface TotalOrder extends Reliable {}
public interface FIFOOrder extends Reliable {}
public interface CausalOrder extends FIFOOrder{}
public interface Timely extends Obvent {
  public long getTimeToLive();
  public long getBirth();
}
public interface Prioritary extends Obvent {
  public int getPriority();
}

/* exceptions */

public abstract class NotificationException
  extends Exception {...}
public class CannotPublishException
  extends NotificationException {...}
public class CannotSubscribeException
  extends NotificationException {...}
public class CannotUnsubscribeException
  extends NotificationException {...}

/* subscription handle */

public final class Subscription {
  public void activate() throws CannotSubscribeException;
  public void activate(long id)
    throws CannotSubscribeException;
  public void deactivate()
    throws CannotUnsubscribeException;
  public void setSingleThreading();
  public void setMultiThreading(int maxNb);
  ...
}
```

Figure 3. Obvents, exceptions, and subscriptions

3.1.2 Obvent Semantics

Obvents can also be viewed as reified messages, or message objects. According to the different semantics that such messages can manifest, several semantics are imaginable for obvents. The first kind of characteristics are the *delivery semantics* associated with obvents; an expression of quality of delivery.

Unreliable: When such an obvent is published, there is no guarantee that it will be received by any notifiable. There is only a *best-effort* attempt to deliver it. This is assumed by default.

Reliable: Once successfully published, a reliable obvent will be received by any notifiable that is "up for long enough". A notifiable which never fails will eventually deliver every such obvent.

Certified: With such obvents, even if a notifiable *temporarily* disconnects or fails, it will eventually deliver the obvent.

Totally ordered: Obvents can furthermore be notified in a total order to the notifiables: roughly spoken, two notifiables `n1` and `n2` which deliver two obvents `o1` and `o2` both deliver `o1` and `o2` in the same order (we also term this *subscriber-side order*).

FIFO ordered: Two obvents `o1` and `o2` that are published through the same object are delivered to all objects whose subscription matches both `o1` and `o2`, and in the same order they were published (*publisher-side order*).

Causally ordered: This type of obvents are delivered in the order they are published, as determined by the happens-before relationship [Lam78]. Note that the notion of event in [Lam78] represents either a message send or receive. These translate respectively to the publishing and receiving of an obvent in our case (*global order*).

Further semantics, called *transmission semantics*, can be associated to obvents. These govern the handling of obvents when they are in transit, with respect to other obvents.

Prioritary: Obvents can have priorities, that is, the delivery of obvents can be delayed to defer to obvents with a higher priority.

Timely: Similarly, obvents can be delayed to prioritize more recent obvents. Also, obvents might expire, and become obsolete.

These different semantics are not all mutually exclusive. For instance, obvents can be certified and have some notion of priority, or be certified and totally ordered at the same time. It appears that contradictions reside for instance between reliable and simultaneously timely limited obvents, as well as between total, fifo or causal order and priorities. In the above cases, the first type takes precedence (Figure 4 illustrates the dependencies between the different semantics). Note however that we have not yet explored all possible ramifications and combinations, and that the identification and implementation of these semantics is an ongoing task.

Note also that for any kind of order expressed by an obvent type, its instances satisfy that order with respect to instances of the same type, its subtypes, and supertypes with that same order only.

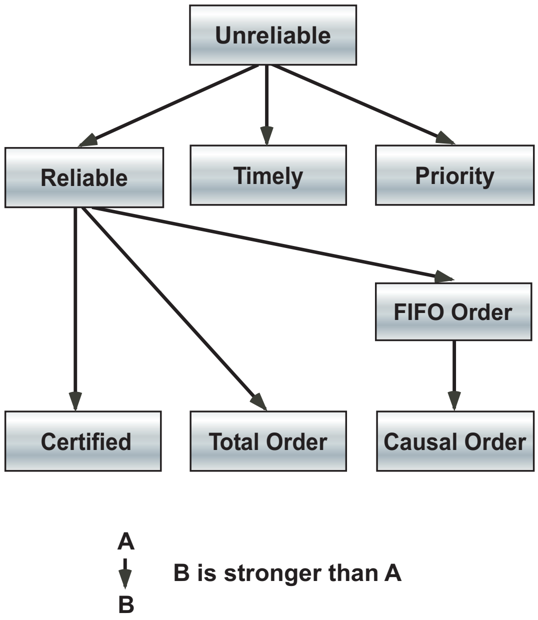

Figure 4. Dependencies between obvent semantics

3.1.3 Expressing Obvent Semantics

In our model, such characteristics are associated with the obvents, and should thus be part of these obvents. Indeed, it makes most sense that every obvent reflects its semantics (which can be seen as a context), such that a correct handling of the obvent can be assured at every moment of the transfer. Since instances of an obvent type are bound to the same obvent source, they present the same characteristics. In addition, the obvent type is the only contract between publishers and subscribers, and we have thus chosen to use subtyping to express this limited form of QoS, mandated by *LP4*. Figure 3 shows the Java types corresponding to the different semantics outlined above.

Since several characteristics can be combined, this scheme requires some mechanism of expressing multiple subtyping.

LM2 *Multiple subtyping.* While simple subtyping eases the expression and addition of different event semantics, multiple subtyping enforces the composition of such semantics.

This is independent of whether it is assured through some form of multiple inheritance as offered by C++, Cecil [Cha95] or Eiffel, through subtyping abstract types (e.g., interfaces in Java), or even *mixins* (e.g., Flavors [Moo86], Ada [Ada95]). The term *multiple subtyping* here simply denotes the ability of expressing multiple specialization relationships.

3.2 The `publish` Primitive

An obvent can be published, which means that it will be asynchronously sent to any concerned notifiable. Following the Java language specification grammar [GJS00], based on a LALR(1) syntax, we introduce a new statement (our specific definitions will be titled in bold font).

PublishStatement:
`publish` *Expression* `;`

Here *Expression* is a non-`null` expression of type `Obvent`, as opposed to most classes relying on Java serialization: in Java, a serializable root type is often "faked" by using formal parameters of the root type `java.lang.Object`, yet expecting an object of a specific type `java.io.Serializable` and throwing an exception if the actual argument is not of that type. We prefer detecting such type errors at compilation. Nevertheless, this primitive can throw an exception of type `CannotPublishException`, signalling any problems in transmitting the obvent. Figure 3 summarizes the basic exceptions.

3.3 The `subscribe` Primitive

As briefly shown in Section 2, we introduce a second primitive `subscribe`, to express a subscription.

3.3.1 Syntax

A subscription expression combines the subscription to a type `T` with (1) a closure declaration representing a filter, where the full signature of that first closure is the following:

```
boolean (T t) {...}
```

and (2) the declaration of a second closure representing the handler, where the full signature is the following:

```
void (T t) {...}
```

A subscription expression hence has the following syntax in Java (details are given in Figure 5):

SubscriptionExpression:
`subscribe (` *ObventType Identifier* `)` *Block Block*

ObventType represents a type which can be widened to the `Obvent` type, that is, *ObventType* is a special case of the *ClassOrInterfaceType* (§ 4.3 in [GJS00]). The filter represented by the first *Block* must return an expression of type `boolean`, while the evaluation block, which we refer to as *event handler* or simply *handler*, returns nothing. The creation of a subscription returns an object of type `Subscription` (cf. Figure 3). Such a handle uniquely identifies a subscription, and is required, for instance, for the activation/deactivation of subscriptions.

SubscriptionExpression:
`subscribe` *SubscriptionDeclaration*

SubscriptionDeclaration:
SubscriptionDeclarator FilterBody HandlerBody

SubscriptionDeclarator:
(*SubscriptionFormalParameter*)

SubscriptionFormalParameter:
ObventType Identifier

FilterBody:
Block

HandlerBody:
Block

ObventType:
ClassOrInterfaceType

Figure 5. Precise syntax of subscription statements

3.3.2 Handlers

Handlers are very close to the closures known form Smalltalk (*block closure*) or Cecil (*anonymous function*), and represent an intuitive way of handling callbacks from the underlying event dissemination system.

In languages like Java, which lack support for closures (or *higher order functions* [OW97]), such callbacks are often implemented by having the application provide a callback object with a callback method. The argument of such a method represents the effective event of interest. In Java, an interface implemented for callbacks is commonly called a *listener*. To enforce strong typing, the type of the formal argument of a callback method in a listener must conform to the type of the event of interest. This can be achieved with *parametric polymorphism* [EG01] (see Section 5), but in any case leads to isolating the event handling in a separate class.

The use of a closure on the other hand enables the regrouping of all code related to a subscription in a single succinct expression, and since handlers are a specific type of closures, a language which already provides some general form of closures can easily support the integration of subscription expressions.

LM3 *Closures.* With handlers being a specific type of closures, a language which provides general closures supports the expression of safely typed subscription primitives.

By viewing these closures as objects, the handlers take the role of the *notifiables* outlined in Section 2.

3.3.3 Filters

Akin to handlers, filters are closures with a specific signature. Besides the concentrating of subscription-related code, the use of such a syntax in the case of filters is conducted by a further desire, namely the confining of code both for the filtering, and especially "revealing" it. This enables (1) the migration of such code to foreign hosts, as well as (2) the factoring out of redundancies between filters of different subscribers gathered on individual hosts.

The compilation of these filters is hence deferred, in a sense similar to the paradigm of deferred code evalution known from *two-level programming* [NN88] (or generalized to more than two levels, *multi-stage programming* as advocated by MetaML [TS97]).

LM4 *Deferred code evaluation.* A mechanism providing some form of deferred code evaluation can easily support the expression of safely typed content-based filters, in a way that supports optimizations (avoiding redundant queries) and the checking of the code.

3.3.4 Restrictions on Closures

"Local" closures vary in the degree of self-containment they advocate: the first class block closure in Smalltalk can use any variables in scope at the closure declaration (at compilation), and these variables are bound for the entire lifetime of the closure, even if it is executed in a context where some of these variables are not visible. To avoid some of this binding of variables, an anonymous class in Java can only access `final` variables from the enclosing block in addition to the variables (members) in scope at compilation. The handlers described previously adopt these semantics.

Our "distributed" use of closures in the case of filters requires even more restrictions. Any variable used in a filter might reference an object (which might reference an object, etc.) of a type which is not known on a host where that filter is evaluated, forcing the transfer of code. Similarly, any method invocation, whether performed on a variable or as a static call, might force the transfer of further code. In the case of Java, a class can be compiled if the types it uses are present as byte code, and it is very difficult to foresee the effects of calls to classes based on their byte code. Thus, method invocations (including the use of constructors) should be cut down to the essential ones, in order to avoid filters using opaque code and/or types unknown on filtering hosts.

Invocations: The only method invocations allowed in a filter are (nested) invocations on its variables.

Variables: The only variables allowed in a filter are (1) the formal argument representing a filtered obvent, (2) local variables, and (3) `final` outer variables (from the enclosing block or class members). Latter two types of variables are restricted to primitive types (e.g., `int`) and their object-counterparts (e.g., `java.lang.Integer`), including `java.lang.String`.

These restrictions enforce the location-independency of the expressed filter, offering the possibility of applying it at a more favourable stage (e.g., a remote host) to reduce network load and filtering cost. If the filter declares any local variables of unallowed types, or performs invocations differing from the ones described above, its migration might be problematic. In such a scenario, the filter is applied locally (cf. next section).

Note that the above restrictions do not fully guarantee the location-independency of the filter, e.g., any Java object referred to by a variable gives access to a meta-object representing its class (`java.lang.Class`), through which many "undesired" things can be done. We will discuss this in Section 5.

3.3.5 Thread Semantics

Once an obvent has reached a process hosting an interested notifiable, it is delivered by executing the handler. This is comparable to an RPC-style invocation of an arbitrary method of a *bound* object, in the sense that the thread which is used in that target object's process for the invocation is blocked until completion of that invocation; the main difference being that an invocation made in the context of an RPC can yield a reply which is sent back to the invoker. There are different levels of concurrency which can be supported. In our context, we distinguish between two kinds of *thread semantics* (*policies*):

Multi-threading: A handler can be executed concurrently for any number of obvents. These semantics are assumed by default, except in the case of ordered obvents.

Single-threading: A handler never processes more than one obvent at a time.

One could easily extend this set, for instance by a thread policy ensuring that only one instance of the same obvent class is processed at a time.

Note that Java already integrates mechanisms for concurrency control with which the above policies can be achieved. To ensure that never more than one obvent is processed at a time by a handler, one could easily write:

```
final Object lock = new Object();
Subscription s =
  subscribe (T t) {...} {
    synchronized(lock) { /* handler */ }
  };
```

However, in languages which do not integrate any concurrency mechanisms, to obtain more sophisticated concurrency control, or to involve the publish/subscribe system into concurrency issues with the goal of optimizing concurrency, thread policies should be made explicit. In our case, it seems most straightforward to express these through the subscription handle, since such an object uniquely defines a subscription. To control such parameters, corresponding methods are added to the `Subscription` type shown in Figure 3.

In general, the expression of QoS has been probably the most tedious task when devising our language primitives. Such QoS seem to become increasingly important when programming at a distributed scale, but there is to our knowledge only very little work on how to inherently express QoS in a programming language, in an other way than through an API.

3.4 Managing Subscriptions

As explained above, the `subscribe` primitive creates an expression representing a subscription. Such a subscription must then be activated, triggering the effective action of subscribing, and later on, deactivated, representing the action of unsubscribing.

3.4.1 Activating a Subscription

A subscription is activated by a call to the `activate()` method on the corresponding subscription handle. This method throws a `CannotSubscribeException` exception if the subscription can not be issued, e.g., if the subscription is already activated.

```
Subscription s = ...;
s.activate();
```

The variant of the `activate()` method with a `long` argument is used in combination with certified events. Indeed, with such events, the lifetime of subscriptions might exceed the actual lifetime of the hosting process. When recovering from a failure, or reactivating an intentionally deactivated subscription, the concerned subscription can be (locally) uniquely identified by using this method.

3.4.2 Deactiving a Subscription

Similarly the action of unsubscribing is expressed through a `deactivate()` method defined on subscription handles, which can through a `CannotUnsubscribeException`.

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

As an immediate consequence, subscriptions can be cancelled also from inside a subscription, i.e., its associated handler. This is interesting when a particular event, from the point of view of the concerned subscriber, supersedes any following events, or signals the absence of any further events. Since a handler can only handle `final` variables declared in its enclosing block however, the variable that the subscription handle is affected to must be declared outside of that block, for instance as a private attribute of the enclosing class.

The activation/deactivation of subscriptions can be interleavingly performed an unlimited number of times. Corresponding exceptions are also thrown upon an attempt of (de-)activating an already (de-)activated subscription.

4 Implementation Issues

This section depicts how we have implemented our primitives for publish/subscribe interaction in Java in a way that satisfies the principles stated in Section 1.

4.1 General Implementation Choices

Along the lines of extensions to the Java language like Pizza [OW97] (adding parametric polymorphism, algebraic types and closures) or [BC97] (for multimethods), we refrain from incorporating any new features into the Java virtual machine, as well as from extending a given compiler, or even modifying existing packages and classes of the Java environment. Instead, we advocate the use of a precompiler as the publish/subscribe counterpart to the `rmic` compiler for generation of remote invocation proxies [Sun99b].

Since remote invocations benefit from an inherent support from the language, they require no specific primitives, and hence only remotely invocable Java types have to be compiled with `rmic`. Besides generating obvent-specific classes, our `psc` precompiler translates publish/subscribe statements and expressions to calls to these classes, and must hence be run not only on obvent types, but also on any class making use of our primitives.

4.2 DACE Distributed Architecture

The *Distributed Asynchronous Computing Environment* (DACE) infrastructure has been initially developed as a general architecture to support publish/subscribe interaction, and has later been specialized for type-based publish/subscribe. The DACE architecture can be roughly pictured as relying on a *class-based* dissemination [EG01]. Every obvent class is mapped to a dissemination channel, representing a multicast group, which we refer to as *multicast class*. In the DACE architecture, such multicast classes are then implemented with different multicast protocols with guarantees ranging from strong guarantees (exploiting a broad variety of primitives from *group communication* [BJ87], e.g., for causal ordering) to primitives with weaker guarantees but strong focus on scalability (*network-level* protocols like IP multicast [DC90] or its derivates, e.g., [PSLB97, HSC95, FJM+96], or *gossip-based* protocols, e.g., [BHO+99, SS00, EGH+01]).

We have adopted a reflexive approach, by using specific channels to disseminate protocol messages, like subscription/unsubscription requests, or the advertisement of the publishing of obvents. Such messages are obvents themselves, and allow distributed processes to learn about other, possibly new, multicast classes.

4.3 Typed Adapters

To avoid making the Java virtual machine distribution-aware, and also to exploit our class-based dissemination, we adopt the *adapter* [OPSS93] concept. Adapters are intermediate entities between the communication system and the application, whose role consists mainly in mediating between events in a serialized representation and objects.

In our case, adapters mainly mediate between serialized generic objects and strongly typed obvents. In other terms, adapters are type-specific, and are generated for each obvent class by the `psc` compiler. For any given obvent class `C` `psc` generates a class `CAdapter` with code for publishing/subscribing instances of `C`. Similarly, to support subscriptions to abstract types (interfaces), for any given abstract obvent type `I`, `psc` generates a class `IAdapter` with code for subscribing to instances of `I`. Figure 6 illustrates an adapter for a given obvent type `T`.[3]

[3] Interestingly, the same parametric polymorphism applied to the first class adapters used in our library approach [EGS00], can not be applied here due to the purely static nature of the adapters.

```
import java.pubsub.*;

public final class TAdapter {

  public static Subscription subscribe(LocalFilter l,
                                       Notifiable n)
    throws CannotSubscribeException {...}
  public static Subscription subscribe(RemoteFilter r,
                                       Notifiable n)
    throws CannotSubscribeException {...}

  /* if T is a class */

  public static void publish(T t)
    throws CannotPublishException {...}
  ...
}
```

Figure 6. Obvent adapter for a type `T`

4.4 Translating Primitives

With our `psc` precompiler, publish/subscribe statements and expressions are translated to method invocations.

4.4.1 Publishing

Since a published obvent is disseminated through the adapter for its dynamic type, which is only known at runtime, a *PublishStatement* can not be directly transformed to a call to `publish` on the corresponding adapter class. Hence, we add a `publish()` method to the `Obvent` interface in Java (Figure 7), whose body is however automatically generated by `psc` for each obvent class `C`:

```
public class C ... {
  /* generated by psc */
  public void publish()
    throws CannotPublishException
    { CAdapter.publish(this); }
  ...
}
```

Accordingly, a *PublishStatement* expressing the publishing of an `Obvent` `o`,

```
publish o;
```

is transformed into a call to the `publish()` method of `o`, provided that `o`'s static type can be widened to `Obvent`.

```
o.publish();
```

4.4.2 Subscriptions

By similarly transforming subscriptions to (static) calls to the corresponding obvent types, subscriptions

```
package java.pubsub;

import java.io.*;

/* obvents */

public interface Obvent extends Serializable {
  /* generated by psc */
  public void publish() throws CannotPublishException;
}

/* top level */

public final class ObventAdapter {...}
  public static Subscription subscribe(LocalFilter l,
                                       Notifiable n)
    throws CannotSubscribeException {...}
  public static Subscription subscribe(RemoteFilter r,
                                       Notifiable n)
    throws CannotSubscribeException {...}
}

/* filters */

public interface Filter {...}
public interface RemoteFilter extends Filter {...}
public interface LocalFilter extends Filter {
  public boolean eval(Obvent o);
}

/* handlers */

public interface Notifiable {
  public void notify(Obvent o);
}
```

Figure 7. Details on `Obvent`**,** `Subscription` **and further types in** `java.pubsub`

to interfaces would be impossible. Hence, subscriptions, as well as unsubscriptions, are handled differently. In short, a subscription statement involving a type `T` is transformed to an invocation of one of the `subscribe()` methods in class `TAdapter` (possibly also `OventAdapter` in package `java.pubsub`), as shown in Figure 6.

An instance of an anonymous class representing a notifiable, is created from the handler of a subscription expression such as the following:

```
subscribe (T t) {...} { /* handler */ }
```

It implements the `Notifiable` interface given in Figure 7:

```
new Notifiable() {
  public void notify(Obvent o) {
    T t = (T)o;
    /* handler */
  }
}
```

Such an anonymous class declaration represents an expression, and can thus be passed as argument to the `subscribe()` method of the corresponding adapter.

4.4.3 Filters

The handling of filters represents the most complex task during precompilation. A filter whose statements deviate from the guidelines which strongly enforce its mobility according to the previous section, is similarly transformed into an anonymous class representing a unary predicate of type `LocalFilter` shown in Figure 7, and applied locally. A subscription expression such as

```
subscribe (T t) { /* filter */ } {...}
```

is hence transformed into an invocation of the corresponding adapter class:

```
TAdapter.subscribe(
  new LocalFilter() {
    public boolean eval(Obvent o) {
      T t = (T)o;
      /* filter */
    }
  },
  new Notifiable() {...}
)
```

If the depicted restrictions are respected, `psc` generates an intermediate representation of the filter, in a way similar to what is done for *application-specific handlers* (ASHs) [EWM96], low-level message filters, except that those are applied locally and expressed in a neutral specification language, while our filters promote the use of the native language syntax. Our precompiler generates two tree-like constructs, which are more specific than for instance the parse trees used in Smalltalk [Riv96].

Invocation tree: First, a representation of the invocations made in the filter is generated: the root represents the filtered obvent, and every node represents a method invocation. A leaf node stands for the outcome of a condition on the value obtained by applying the methods of the nodes on the path downto that leaf in a nested fashion (nodes can also represent attribute accesses).

Evaluation tree: Second, a tree representing the relationships between the leaves of the former tree and the outcome of the filtering is generated: its nodes represent mainly logical combinations of its subnodes aso., and the leaves are references to the leaves of the former tree.

This information is stored in an instance of `RemoteFilter` (Figure 7).

A general description of our approach to instrumenting Java with first class parse trees, based on a generalization of deferred code evaluation expressed through filters, will be the subject of a future paper.

5 Discussion

This section discusses several issues, including two design alternatives as well as interoperability issues. We also contrast our type-based publish/subscribe with RMI, pointing our the fact that the two paradigms are not contradictory but complementary.

5.1 Fork

We have explored several alternative primitives for the expression of publish/subscribe based on our obvent model, among which we outline what we believe to be the two which are most likely to come into mind. This first alternative for the expression of publish/subscribe based on our obvent model makes use of a `fork`-similar primitive for notification delivery.

5.1.1 Obvent Variable

A new notification is assigned to a variable, and a block representing a handler is executed every time a new value is put into the variable:

```
T t = null;
t = subscribe {...} {...};
/* here t is null */
```

This primitive could, in the case of Java, be implemented similarly to the solution presented throughout this paper. However this syntax makes it difficult to express unsubscriptions, a problem which does not occur in the case of the `fork` primitive: when spawning a new coroutine, the execution of the corresponding block takes place once only. Here, notifications are delivered continuously, leading to a repeated evaluation of the handler. By the absence of a subscription handle, a subscription can not be referred to from outside of its expression. Unsubscriptions would have to be dealt with inside the handlers, either through a parameterless unsubscription statement, or by having the handler return a boolean value after each obvent evaluation to signal whether the subscription is to be pursued. Either variant leads to a restrictive solution, where a subscription can *only* be cancelled after the next event has been delivered. While this can be desirable in many case, which has motivated the support included in the solution presented throughout this paper, it should not be the only possibility of cancelling a subscription.

5.1.2 Filter

Note that alternatively, with this model, one could think of implementing filters by giving the application the possibility of instantiating the future variable `t`, and in that sense, using it simultaneously as a *template object*:

```
T t = new T(...);
t = subscribe {...};
```

However, filtering events by matching them against template objects offers only little expressiveness: template objects are usually compared attribute-wise with events, making the matching of an attribute against a range of values, or, since attributes might themselves be objects, the matching only of a nested attribute difficult to express. Alternatively, the matching can be performed inside the template object, which renders the matching opaque, disabling any optimizations targeted at avoiding redundancies.

5.2 Callback

As outlined in Section 3, a very common way of implementing a callback in a language such as Java consists in asking the application to provide a callback object implementing a given interface.

5.2.1 Listener

This second considered alternative, which is promoted by nearly all Java API's for common publish/subscribe engines, introduces a specific listener like the `Notifiable` type presented in Figure 7. This type, which, as depicted, is inherently implemented by every anonymous class representing a handler, would in this case be explicitly implemented and instantiated by the application to catch event notifications, e.g.:

```
Notifiable n = ...;
Subscription s =
  subscribe (T t) { / *filter* / } n;
```

The declared `notify()` method with its weakly typed argument does however not address *LP1*. A *dynamic overriding* of the `notify()` method, i.e., adding a variant with an argument type `T` that subtypes `Obvent` in order to subscribe to instances of `T`

```
public void notify(T t) {...}
```

would also allow handlers for several obvent types to be provided by the same notifiable, yet does not add type safety.

5.2.2 Dispatching

Through this dynamic overriding, the `notify` method becomes a so-called *multi-method*. Unfortunately, *dispatching* (method selection) in Java, unlike CLOS [Jr.90] or Cecil, does not support multi-methods. Dispatching in Java, similarly than in C++, offers *dynamic uni-dispatch*, i.e., the class of the object referenced by an invoked variable (representing its dynamic type) is determined at runtime, but only *static multi-dispatch*. This prevents a typed solution based on dynamic overriding of the `notify()` method desribed above.

There have been several approaches to overcoming Java's lack for multi-methods, ranging from using reflection over modifying compilation to extending the virtual machine. Another common technique is also given by *double dispatch* [Ing86], which could however not be applied in this case, like some of the above solutions, unless a specific notifiable `TNotifiable` is generated and implemented for every obvent type `T` that a notifiable type handles. This would however remove the main advantage of this approach based on listeners, which consists namely in giving much freedom to the developer in defining event handlers for any types (including subtypes of subscribed types). In addition, the required dynamic multi-dispatch is a high price to pay, especially if only introduced for the use with publish/subscribe primitives, also given the fact that it does not improve type safety.

Note furthermore, that the scenario of multiple subscriptions of a notifiable to the same type, or through subtyping related types, is not straightforward to handle: are the different filters combined, and is the same event delivered several times? On the other hand, the callback approach also enables an easy expressing of thread semantics: by subtyping a `SingleThreadNotifiable`, the developer could express the desire of processing only one instance of a given obvent type at a time. Multiple subtyping, introduced to express a limited form of QoS for obvents, could here be used again to express complex event handling semantics.

5.3 Obvents vs Objects

In languages which provide a default serialization mechanism through a type which must be subtyped, e.g., Java (`java.io.Serializable`), obvents are instances of that type. Hence in a general case, as shown by our distinction between bound/unbound objects, not every object can be an obvent: the opposite would have not only mandated the type system to be singly-rooted (requiring extensions to languages like C++ and Ada), but also to integrate the before-mentioned serialization mechanism at that very root like in Smalltalk, possibly including a `publish()` method.

How about an obvent publishing obvents, or subscribing to obvents? The former case does not bear any particular dangers, and should thus not be prohibited. Similarly, subscriptions could also be issued inside obvents, since a subscription does not affect the object in which it was performed. Subscriptions only have a "local" effect, since an instantiated closure can not be referenced. Hence the effective handlers/notifiables can not be passed around, and in particular, can not be attributes of obvents. To underline this characteristic of subscriptions, the `Subscription` type is not serializable.

Note that in the second presented design alternative, any application object can be defined as a notifiable by implementing a given interface, that is, a `notify()` method. Equipping all objects with the ability of subscribing would have translated to adding that `notify()` method to the root type. Furthermore, notifiables could be attributes of obvents. The danger of endless recursions whenever an obvent subscribes to itself could however easily be banned, by not considering a subscription as part of the state of a subscribed obvent: once published, the copies of an obvent would have to reissue any subscriptions performed by their "master copy".

5.4 RMI and Publish/Subscribe

Originally introduced as *remote procedure call* (RPC) [BN84], remote invocations have been quickly applied to object-oriented languages, leveraging some form of remotely accessible entity, e.g., *guardians* in Argus [Lis88] (its follow-up CLU [Lis93]), *network objects* in Modula-3 [CDJ+89] and Obliq [Car95] (every object is potentially a network object), and of course *remote objects* in Java RMI [Sun99b].

5.4.1 Invocations vs Events

There are mainly two differences between such remote invocations and our event-based model:

Interaction styles: The RPC model promotes the same abstraction for remote object interactions as for local ones. By doing so, (synchronous) RPC is inherently integrated with the language, and requires little more support than that very inherent interaction abstraction. In contrast, our model promotes two interaction styles, namely (1) event-based publish/subscribe interaction remotely, and (2) method invocations locally, making the application developer more aware of distribution.

Object passing semantics: With remote invocations, there are two possible ways of passing objects, namely (1) by *reference*, i.e., a proxy object is created in the receiver's address space, and (2) by *value*. The distinction goes along the notion of the granularity mediated in Section 2, i.e., "large" locality-bound objects interact via remote invocations, where the invocation arguments are "small" unbound objects or references to "large" remotely invocable objects. In contrast, when using our obvent-based model, objects are primarily passed by value.

However, publish/subscribe and RPC are not contradictory, but complement each other. A combination of both represents a very powerful tool for devising distributed applications, e.g., by passing object references with obvents.

5.4.2 Hand in Hand

A collaboration between the two interaction styles can be illustrated by reconsidering the simple example given in Section 2. Though the use of a publish/subscribe interaction for the dissemination of stock quotes seems appropriate by scaling easily to many brokers, it might seem more appropriate to use a sychronous interaction with the stock market when purchasing stock options, e.g., a remote method invocation.

Figure 8 shows how Java RMI and publish/subscribe can work together, hand in hand. To correctly handle the serialization of remote objects (in the sense of RMI), the RMI serialization mechanism is used, enabling a transparent integration of RMI and publish/subscribe. However, the current implementation of Java RMI presents a severe caveat, which becomes especially visible through this integration with events. In fact, the distributed garbage collection keeps a remotely accessible object from being garbage collected as long as there is at least one proxy for that object. When publishing an event containing a reference to a remote object, such a proxy is created for each subscriber, which can sum up to several 1000's. Every time a proxy is garbage collected, the Java virtual machine hosting the represented object is notified. Consequently, if a single subscriber crashes, the remote object will never be garbage collected. With a "weaker" implementation of Java RMI, such as the one proposed in [CNH99], this problem could be circumvented.

5.5 100% Pure Content

Java, like most statically typed languages, uses *name equivalence* of types, which means that two types are compatible only if declared so. If two different types `T1` and `T2` have exactly the same parents in their type hierarchy, instances of `T1` can not be assigned to variables of `T2` or vice versa. A type scheme which allows this is said to enforce *structural equivalence* of types.

5.5.1 Reflection

When subscribing to obvents it might be of interest in certain cases to subscribe to *any* obvents which implement a given method (specified by its name and formal/actual parameters) irrespective of the types of these obvents. Java, just like Smalltalk, provides *introspection* mechanisms [Sun99a], that enable the querying of objects for their type, and also members (attributes and methods, including facilities for dynamically reading/writing former ones and invoking latter ones). Every Java object gives access to these features through a method `getClass()`, which would enable the expression of a purely content-based subscription like the following:

```
Subscription s =
  subscribe (Obvent o) {
    ...
    Class c = o.getClass();
    Method m = c.getMethod("getPrice", null);
    return
      m.invoke(o,null).equals(new Float(150));
    ...
  } {...};
```

Now, *any* obvent type which implements a given method `getPrice()` (e.g., the `StockQuote` class) could

```
import java.rmi.*;
import java.rmi.server.*;
import java.pubsub.*;

/* user */

public interface StockBroker extends Remote {...}

/* stock market */

public interface StockMarket extends Remote {
  public boolean buy(String company, float price,
                     int amount, StockBroker buyer)
    throws RemoteException;
  ...
}

/* stock quote obvents */

public class StockObvent implements Obvent {
  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 StockObvent(String company, float price,
                     int amount)
  {
    this.company = company;
    this.price = price;
    this.amount = amount;
  }
}
public class StockQuote extends StockObvent {
  private StockMarket market;
  public Stockmarket getMarket() { return market; }
  public StockObvent(String company, float price,
                     int amount, StockMarket market)
  {
    super(company, price, amount);
    this.market = market;
  }
}

/* subscribing */

final StockBroker broker = ...; /* this broker */
...
Subscription s =
  subscribe  (StockQuote q) {
    return (q.getPrice() < 100 &&
      q.getCompany().indexOf("Telco") != -1);
  }
  {
    ...
    boolean bought =
      q.getMarket().buy(q.getCompany(), q.getPrice(),
                        q.getAmount(), broker);
    ...
  };
...
```

Figure 8. Buying stock quotes

be captured by this filter, and the following handler could similarly dynamically extract information from a conforming event. Note that this feature, though giving much flexibility to the application developer, is somehow opposed to our requirement for type safety (*LP1*), and we therefore do not consider introspection as a necessary language mechanism, though our current Java prototype supports such untyped filters. Note that this scheme is usually realized by using specific event types implementing introspective mechanisms, like the *self-describing* messages advocated by [OPSS93].

5.5.2 Tuples: Back to the Roots

Another way of achieving structural equivalence could consist in coming back to the concept of *tuples* used in Linda [Gel85] (see next section), one of the spiritual ancestors of the publish/subscribe pardigm. In that sense, the `publish` primitive could be extended in order to accept any number of actual arguments:

```
String company = ...;
float price = ...;
int amount = ...;
StockMarket market = ...;
publish (company, price, amount, market);
```

Inversely, the `subscribe` primitive could be used with an arbitrary number of formal arguments, e.g,

```
Subscription s =
  subscribe (String company,
             float price,
             int amount,
             StockMarket market)
  { /* filter */ }
  { /* handler */ };
```

which could all be used as subscription criteria by the filter, and could all be accessed by the handler.

Though one can argue that languages which do not inherently support structural equivalence should not be instrumented with a distributed interaction style that relies on that paradigm, we believe that this could lead to a very appealing style of distributed programming, but requires a more complex filtering.

5.6 Language Integration vs Interoperability

Language integration seems to be somehow contradictory with the requirement of interoperability expressed in [OPSS93]. Publish/subscribe has however nowadays found application in various contexts (e.g, virtual reality [HMN+00], group collaboration [MHJ+95], real time [Fet98]), mainly due to its scalability properties achieved through strong decoupling of participants.

Just like CORBA [OMG01a] has added interoperability to the RPC through a neutral *interface description language* (IDL), there have been several approaches to using a neutral *event description language* (EDL) to help adding interoperability to publish/subscribe. Proposals for such languages are plentiful, like the OMG's *object definition language* (ODL) used in the Cambridge Event Architecture [BMB+00], languages used in tuple space implementations, e.g., *object interchange language* (OIL) in Objective Linda [Kie95], or more recently, XML.

We believe that such a specification language could help adding interoperability to our system, and that in that sense, the types in package `java.pubsub` can be seen as a Java mapping. However, since events are in our case not pictured as structures as in the above cases, but also encompass methods and thus code, an EDL can not by itself provide for interoperability. The issue of passing objects by value from one language to another has also been tackled in CORBA, and we are currently investigating on the impact of an application of those concepts to our system.

6 Related Work

Though publish/subscribe engines are plentiful, there has been only little work on integrating publish/subscribe interaction with an object-oriented programming language up to now. We first overview the two closest efforts, and then focus on linguistic support for alternative distributed interaction styles other than RPC, which has already been discussed previously.

6.1 Publish/Subscribe

Most event-based solutions focus less on language integration, and more on interoperability.

6.1.1 ECO

One approach to integrating event-based interaction with C++ is discussed in [HMN+00], but also starting from a general event model. The authors dissociate two ways of adding event semantics to an object-oriented language, namely (1) by extension with specific constructs and (2) by addition of specific classes, adopting the first approach. Their ECO (*events + constraints + objects*) model incorporates events as specific language constructs which are complementary to objects, and thus necessitate a considerable number of language add-ons. The use of a precompiler to handle these extensions is also mentioned, but [HMN+00] gives no details about its implementation.

6.1.2 CEA

The Cambridge Event Architecture (CEA) [BMB+00] promotes a *publish/register/notify* interaction style, where an intermediate event trader mediates between publishers and subscribers. The CEA is based on an interopable object model, in which events are described by the ODMG's *object definition language* (ODL), but alternative specification languages, like XML, are also mentioned. Events are typed according to the definition language, and C++ and Java mappings are mentioned. Filtering mechanisms are also included, however based on viewing the events as sets of attributes, forcing the application to define filters based on attribute-value pairs.

6.2 Message Passing

Language integration has evolved along the communication paradigms. One of the first distributed interaction paradigms was *message passing*, consisting basically of a usually asynchronous *send* of messages (mainly for single sends, but also for addressing groups similar to `publish`), and a blocking *receive* primitive. Like many "distributed" interaction paradigms, message passing has also extensively been used first in parallel and concurrent programming. Early, mainly procedural and process-oriented languages, like Occam [Pou84] (using named unidirectional channels) or also SR [AO86] (first only with *semi-synchronous* sends, later also with synchronous remote procedure calls), included some form of message passing.

6.3 Tuple Spaces

The well-known *tuple space* paradigm [Gel85] represents the distributed interaction scheme which is closest to publish/subscribe.

6.3.1 The Original

The tuple space, first introduced for parallel computing in Linda, can be viewed as a form of distributed shared memory, with the main difference being the structured form of data inserted into and retrieved from the tuple space: *tuples* inserted into a tuple space are sets of values, and tuples read from the space are specified as a set of formal and actual arguments, where former ones can be seen as placeholders defining the types of the corresponding elements, and latter ones define the values of the corresponding elements for a candidate tuple.

6.3.2 Decoupling

The original tuple space in Linda was based on a form of name-based addressing, i.e., tuple elements could be strings carrying a specific name. This idea has been reused in the "original" publish/subscribe variant based on topics, and the possibility of using more than one name element in Linda has been straightforwardly transposed to topic hierarchies, expressing containment relationships on topics. In general, the publish/subscribe paradigm has been strongly influenced by the tuple space, more specifically by its strong *decoupling* of participants in *time* and in *space*.

6.3.3 Publish/Subscribe vs Tuple Space

Publish/subscribe further increases this decoupling by adding flow decoupling, that is, replacing the synchronous interaction (*pulling*) of the subscribers with an asynchronous notification mechanism. In fact, the original tuple space had three primitives, namely (1) `out` to push a tuple into the space (similarly to `publish`), (2) `read` to read a tuple without erasing it, and (3) `in` to withdraw a tuple from the space. Publish/subcribe adds an asynchronous callback mechanism, and by omitting an equivalence to `in`, sacrifices concurrency control to scalability.

6.3.4 Linguistic Support

There have been a series of attempts to transform the structured form of tuples to an *object* form, mainly by extending the exact type equivalence for tuple elements in Linda to the notion of subtyping. While early approaches to integrating the tuple space with an object language like [MK88] (for Smalltalk) promoted tuples as sets of objects, later approaches, like [Pol93] (C++) or [Kie95] (Objective Linda) considered tuples, just like events in our case, as single first class citizens, and added some form of content-based matching based on templates. Very recently, callback mechanisms have also been added, e.g. JavaSpaces [FHA99], TSpaces [LLW99] (both Java), supporting a publish/subscribe-like interaction. In contrast to our approach however, latter ones basically promote publish/subscribe interaction through some weakly typed reified bus, and advocate template-based matching.

7 Concluding Remark

Throughout the history of computing, many paradigms have first been implemented as separate libraries, and have then made their way into programming languages. The *monitor* abstraction for concurrent programming introduced by Hoare [Hoa74] for instance, was first implemented through a library before becoming a specific language construct (e.g., Portal [Bus88]). Nowadays, every Java object is potentially a monitor.

This evolution does however not apply to all paradigms, as can be shown again by Java, which currently only includes a limited form of *pointers* through a library (`Reference` in `java.lang.ref`), while one of main spiritual ancestors, C++, integrates pointers as a first class concept.

This paper does not aim at advocating an integration vs a library approach to support publish/subscribe programming. Yet, we hope that based on the practical experience we are currently gathering with both our library approach [EGS00, EG01] and the present integration approach, we will be able to answer the question of which kind of approach is more adequate to achieve "object-oriented publish/subscribe".

8 Acknowledgments

Many ideas presented here have been refined through discussions with Andrew Black (Oregon Graduate Institute of Science and Technology), Joe Sventek (Agilent Laboratories, Edinburgh), Martin Odersky, Mathias Zenger, and Michel Schinz (Swiss Federal Institute of Technology, Lausanne), and Nicolas Ricci (Lombard & Odier, Geneva). We would like to thank all the above-mentioned, as well as the anonymous reviewers, for their valuable comments and suggestions.

References

[Ada95] International Organization for Standardization. *Ada 95 Reference Manual - The Language - The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[AEM99] M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS '99)*, pages 43–53, October 1999.

[AO86] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *Distributed Computing*, 1(2), April 1986.

[ASS+99] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, November 1999.

[BC97] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 66–76, October 1997.

[BHO+99] K.P. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[BJ87] K.P. Birman and T.A Joseph. Reliable communication in presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMB+00] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, March 2000.

[BN84] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bus88] A. Businger. *PORTAL Language Description*. Number 198 in LNCS. Springer-Verlag, 1988.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 286–297, 1995.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.

[CDJ+89] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Conference Record of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*, pages 202–212, Austin, Texas, January 1989.

[Cha95] C. Chambers. The Cecil language specification and rationale: Version 2.0. Technical Report UW-CS Technical Report 93-03-05, Department of Computer Science and Engineering, University of Washington, December 1995.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *Conference Record of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*, pages 125–135, 1990.

[CNF98] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '98)*, pages 261–270, april 1998.

[CNH99] M. Philippsen Ch. Nester and B. Haumacher. A more efficient rmi for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, Palo Alto, CA USA, June 1999.

[Cor99] Talarian Corporation. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper).* http://www.talarian.com/, 1999.

[DC90] S. Deering and D. Cheriton. Multicasting routing in datagram internetworks and extended LANs. *ACM Trans. on Computer Systems*, 8(2):85–110, May 1990.

[EBG01] P.Th. Eugster, R. Boichat, and R. Guerraoui. Effective multicast programming in large scale distributed systems. Technical Report DSC/2001/003, to appear in Concurrency: Practice & Experience, Wiley & sons., Swiss Federal Institute of Technology, Lausanne, http://dscwww.epfl.ch/EN/publications/, January 2001.

[EG01] P.Th. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 131–146, January 2001.

[EGH+01] P.Th. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. Technical Report DSC/2001/002, Swiss Federal Institute of Technology, Lausanne, http://dscwww.epfl.ch/EN/publications/, January 2001.

[EGS00] 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.

[ES92] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1992.

[EWM96] D.R. Engler, D.A. Wallach, and M.F.Kaashoek. Design and implementation of a modular, flexible, and fast system for dynamic protocol composition. Technical Report TM-552, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1996.

[Fet98] C. Fetzer. Fail-aware publish/subscribe in Erlang. In *Proceedings of the Fourth International Erlang User Conference*, September 1998.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, June 1999.

[FJM+96] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, November 1996.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[GJS00] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[HBS98] M. Happner, R. Burridge, and R. Sharma. Java Message Service. Technical report, Sun Microsystems Inc., October 1998.

[HMN+00] 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.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[HSC95] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of the 1995 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*, pages 328–341, August 1995.

[Ing86] D.H.H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 347–349, 1986.

[Jr.90] G.L. Steele Jr. *CommonLisp the Language.* Digital Press, second edition, 1990.

[Kie95] Th. Kielmann. Object-oriented distributed programming with objective linda. In *Proceedings of the First International Workshop on High Speed Networks and Open Distributed Platforms*, June 1995.

[Koe99] P. Koenig. Messages vs. objects for application integration. *Distributed Computing*, 2(3):44–45, April 1999.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lis93] B. Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, March 1993.

[LLW99] T.J. Lehman, S.W. Mac Laughry, and P. Wyckoff. Tspaces: The next wave. In *Proc. of Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice-Hall, 1992.

[MHJ+95] A.G. Mathur, R.W. Hall, F. Jahanian, A. Prakash, and C. Rasmussen. The publish/subscribe paradigm for scalable group collaboration systems. Technical Report CSE-TR-270-95, University of Michigan, EECS Department, 1995.

[MK88] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 276–284, 1988.

[Moo86] D. A. Moon. Object-oriented programming with Flavors. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 1–8, November 1986.

[NN88] F. Nielson and H.R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[OMG00] OMG. *Notification Service Standalone Document*. OMG, June 2000.

[OMG01a] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, February 2001.

[OMG01b] OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service*. OMG, March 2001.

[OPSS93] 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.

[OW97] M. Odersky and Ph. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 146–159, Paris, France, 15–17 January 1997.

[Pol93] A. Polze. Using the object space: A distributed parallel make. In *Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 234–239(6), September 1993.

[Pou84] D. Pountain. The Transputer and its special language, Occam. *Byte Magazine*, 9(8):361–366, August 1984.

[PSLB97] S. Paul, K.K. Sabnani, J.C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[Riv96] F. Rivard. Smalltalk : a Reflective Language. In *Proceedings of the International Conference on Metalevel Architectures and Reflection (Reflection'96)*, pages 21–38, April 1996.

[SA97] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG '97)*, http://www.dtsc.edu.au/, September 1997.

[Ske98] D. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*. http://www.vitria.com, 1998.

[SS00] Q. Sun and D.C. Sturman. A gossip-based reliable multicast for large-scale high-throughput applications. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN2000)*, New York, USA, July 2000.

[Sun99a] Sun. *Java Core Reflection API and Specification*, 1999.

[Sun99b] Sun. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*, 1999.

[TIB99] TIBCO. *TIB/Rendezvous White Paper*. http://www.rv.tibco.com/, 1999.

[TS97] W. Taha and T. Sheard. Multi-stage programming. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 321–321, June 1997.

[YBS86] A. Yonezawa, J. P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 258–268, September 1986.