Type-Based Publish/Subscribe: Concepts and Experiences

PATRICK EUGSTER
Purdue University

A continuously increasing number of interconnected computer devices makes the requirement for programming abstractions for remote one-to-many interaction yet more stringent. The publish/subscribe paradigm has been advocated as a candidate abstraction for such one-to-many interaction at large scale. Common practices in publish/subscribe, however, include low-level abstractions which hardly leverage type safety, and provide only poor support for object encapsulation. This tends to put additional burden on software developers; guarantees such as the aforementioned type safety and object encapsulation become of increasing importance with an accrued number of software components, which modern applications also involve, besides an increasing number of hardware components.

Type-based publish/subscribe (TPS) is a high-level variant of the publish/subscribe paradigm which aims precisely at providing guarantees such as type safety and encapsulation. We present the rationale and principles underlying TPS, as well as two implementations in Java: the first based on a specific extension of the Java language, and a second novel implementation making use of recent general-purpose features of Java, such as generics and behavioral reflection. We compare the two approaches, thereby evaluating the aforementioned features—as well as additional features which have been included in the most recent Java 1.5 release—in the context of distributed and concurrent programming. We discuss the benefits of alternative programming languages and features for implementing TPS. By revisiting alternative abstractions for distributed programming, including "classic" and recent ones, we extend our investigations to programming language support for distributed programming in general, pointing out that overall, the support in current mainstream programming languages is still insufficient.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*

General Terms: Languages, Design

This work was supported in part by Agilent Laboratories, Lombard Odier Darier Hentsch and Co., the Swiss Group for Object-Oriented Programming, and the Swiss National Science Foundation. P. Eugster was formerly associated with Sun Microsystems, Inc. and the Swiss Federal Institute of Technology in Lausanne.
Author's address: P. Eugster, Department of Computer Science, Purdue University, West Lafayette, IN 47907; email: p@cs.purdue.edu.

Additional Key Words and Phrases: Abstraction, generics, Java, publish/subscribe, reflection, type, distribution

ACM Reference Format:

Eugster, P. 2007. Type-based publish/subscribe: Concepts and experiences. ACM Trans. Program. Lang. Syst. 29, 1, Article 6 (January 2007), 50 pages. DOI = 10.1145/1180475.1180481 http://doi.acm.org/10.1145/1180475.1180481.

1. INTRODUCTION

When programming distributed applications—and nowadays, virtually any application of industrial scale involves physically separated components—communication abstractions supporting other than strict pairwise ("one-to-one") interaction have proven extremely useful. Indeed, there is an unlimited number of scenarios where we can identify interaction patterns involving components that produce data which is of interest for several consumers. This is typical of multicast-style ("one-to-many") interaction.

One-to-many is not many one-to-one. The scalability requirements posed by modern distributed applications, culminating in *peer-to-peer* and *grid* computing, mandate specific support for multicast and broadcast communication. Building corresponding primitives on top of high-level abstractions for one-to-one interaction such as derivates of the *remote procedure call* (RPC) would clearly hamper the efficiency of these primitives. The increasing system scale of the aforementioned settings furthermore leads to an increased dynamism, which is possibly exacerbated by supporting *mobile* participants. These constraints already make it unfeasible for data producers to know and keep track of all potential consumers. An adequate abstraction imlemented by a dedicated middleware substrate is needed to deal with these issues.

The shooting star publish/subscribe. An abstraction which has proven its value for providing such one-to-many interaction is the *publish/subscribe* [Oki et al. 1993] paradigm. Consumers subscribe to whatever "kind" of *events* they are interested in. Producers anonymously[1] and asynchronously publish events, which are then notified to consumers with matching subscriptions, by sending them the data associated with these events. Many systems provide such event-based interaction models, and implement one or several specifications out of an established large family (e.g., Happner et al. [2002], Freeman et al. [1999] for Java, OMG [2000, 2001b, 2003] for CORBA [OMG 2002]; see Section 8).

Having emerged from *group communication* [Powell 1996], it is not surprising that seminal publish/subscribe systems promoted subscriptions based on *topics* (or *subjects*), which roughly represent groups of participants with common interests. For example, the topic "StockQuotes" regroups all stock brokers interested in stock quotes. These topics were then augmented by inclusion relationships, leading to topic *hierarchies* (e.g., TIBCO [1999], Talarian Corp. [1999], and Altherr et al. [1999]). A consumer subscribing to a topic

[1]This constitutes the main difference to the observer design pattern [Gamma et al. 1995] (which publish/subscribe is often confused with) from an abstraction point of view, which, however, results in a sensible difference in terms of scalability.

"/StockQuotes" hence expresses interest in events published under this topic, and under any subtopic thereof, for example, "/StockQuotes/Telco." Yet more expressiveness was then achieved by taking event *content* into account when subscribing to, and subsequently routing, events. Seminal work includes Carzaniga et al. [2000] and Aguilera et al. [1999]. Corresponding content-based subscriptions then also encompassed content *filters*, some form of predicates that are expressed on the content of (the data associated with) events, such as "all stock quotes which cost less than $100." Most topic-based systems have been augmented with content filtering capabilities.

Implementing publish/subscribe. To disseminate events both reliably and efficiently in large-scale distributed systems manifesting network and process failures, most publish/subscribe engines employ *overlay networks* [Carzaniga et al. 2000; Aguilera et al. 1999]. To fully exploit these infrastructures, events are viewed as plain structures composed of event *properties*, rather than as objects, and subscriptions are expressed in some SQL-like grammar based on these properties. Most of these systems only support predefined event types which represent *maps* of property-value pairs. When put to work in object settings, such systems offer no support for the encapsulation of (the state of) events, and do not leverage type safety, since events are viewed as sets of publicly accessible fields—each of an arbitrary type. In addition, through the expression of subscriptions as strings following some SQL-like grammar, it becomes impossible to verify even the invariable parts of these subscriptions at compilation, making the already burdensome development and debugging of distributed applications even harder.

Such practices are motivated by the need for late binding [Oki et al. 1993], and efficiency: As observed by Carzaniga et al. [2000], routing and filtering infrastructures can hardly scale in the face of overly expressive subscription schemes. Intuitively, if much expressiveness is provided, interests of consumers tend to manifest only little overlap, and an important benefit of overlay networks, namely, the possibility of regrouping common interests [Aguilera and Strom 2000] (and thus common communication), is strongly diminished. Based on such arguments, event-based communication has been claimed to be inherently incompatible with the principles of object-orientation [Koenig 1999].

Type-based publish/subscribe. The goal of type-based publish/subscribe (TPS), our high-level abstaction for multicast interaction, is precisely to provide guarantees such as type safety and encapsulation, without hampering the efficiency of routing and filtering mechanisms.

In short, events in TPS are objects which are instances of "arbitrary" application-defined types. In essence, the TPS paradigm uses an "ordinary" type scheme, without explicitly introducing a topic hierarchy nor any other specific notion of event kind: the type *is* the topic. Effective application events neither have to be explicitly inserted into, or extracted from, any containers or predefined general-purpose types, which greatly leverages type safety. Similarly, consumers neither have to transform nor cast received events or event properties. Furthermore, it turns out that this type information, when provided to a TPS engine, can be employed to set up "connections" (e.g., to preconfigure an overlay network), just like topics in a topic-based system.

Subscriptions in TPS moreover include content filters expressed on the public members of the types of events subscribed to. The content of an event object is hence implicitly defined: the state *is* the content. TPS nevertheless preserves the encapsulation of event objects by not forcing event (types) to reveal their state because content filters can make use of public methods. General-purpose event types such as maps are merely a specific type of event, and can still be used whenever late binding is required, that is, the content of events is unknown at compile-time.

Going the last mile. When going the last mile, that is, when making these concepts of type safety and encapsulation concrete without hampering efficiency of filtering and routing mechanisms, more care is required than when implementing pure content-based schemes [Eugster et al. 2002]. However, by restricting the use of methods in content filters to field access methods, and introducing simple conventions on their naming (possibly with inherent support from the language itself, see [Meyer 1992b]), we can even build TPS on top of prominent content-based engines [Baehni et al. 2002]. The salient difference is that we replace content filters as strings, for example, "'price' < 100.0" (a classical example for SQL-like languages in publish/subscribe systems) by something expressed in the programming language itself, and hence type-checked, for example, e.getPrice()<100.0 (where e is a formal argument representing an event). Through such transformations, ugly aspects of distribution are hidden without introducing a performance penalty, as events can still be efficiently routed and filtered at a low level, that is, without (repetitive) deserialization. Independently of the semantics we support in content filters, the grand challenge of TPS consists in providing a means of expressing subscription criteria, including content filters such as previously described, in the programming language itself, without these being simply compiled and applied locally.

With such an approach, and a set of type conformance rules between languages (à la CORBA), interoperability, a further argument for current practices in addition to efficiency, can also be achieved [Baehni et al. 2003].

Contributions. This article presents the concepts underlying TPS, and experience we have gathered when putting these concepts to work. While we have devised several distributed algorithms for disseminating events (e.g., Eugster et al. [2003], Eugster and Guerraoui [2002], and Baehni et al. [2004]), this article focuses on the last mile. More precisely, the contributions of this work are the following:

—The design principles underlying TPS are presented from a general point of view. These principles are also compared with related work on typed event-based distributed programming schemes.
—We illustrate these principles (in a first step) through an extension of the Java programming language, with two primitives added specifically for TPS programming [Eugster et al. 2001]. This extension approach, which never aimed at a standardization effort, has served as reference for subsequent TPS implementations.
—We present a new "library" prototype, that is, a prototype implemented without any specific language extensions or hooks into the runtime environment.

This approach relies on recent general-purpose features of the Java language [Sun 2005]—generics and (a limited mechanism for behavioral) reflection—which are idealized in a first step for presentation simplicity. This approach supercedes seminal library implementations of TPS [Eugster and Guerraoui 2001; Eugster et al. 2000, 2004; Damm et al. 2004], which were initiated prior to the finalization of the aforementioned features.

—We also evaluate the library approach in more detail by comparing it with the language approach, pointing out weaknesses of the aforementioned features [Damm et al. 2004]. Roughly, generics lack runtime support, and only a limited mechanism for behavioral reflection is available, which is furthermore not supported uniformly. While the first omission has already been criticized in the context of orthogonal persistence (e.g., Solorzano and Alagic [1998]), we present here its ramifications in the context of distributed programming in general by looking also at alternative distributed programming abstractions, such as *remote method invocations* (RMI) and *tuple spaces*. Similarly, limitations of the core mechanism for behavioral reflection adopted by Java have been pointed out earlier (e.g., Liebermann [1986]). We focus here on the impact of the type system on this mechanism and on distributed programming in general, as well as on other new features of the Java language at the 1.5 "Tiger" release [Sun 2005], such as *boxing/unboxing*, or *variable arguments*.

—We then investigate alternative general-purpose language features, in addition to .NET, the main rival of Java, in the context of implementing distributed programming abstractions as libraries, without *specific* support. We point out that in general the support for implementing distributed interaction is insufficient in current mainstream programming environments [Eugster et al. 2004].

Roadmap. The rest of this article is organized as follows. Section 2 presents background information on the publish/subscribe paradigm, and assumptions made. Section 3 introduces the principles underlying TPS. Section 4 presents our TPS-specific extension of the Java language, illustrating the difficulties in putting TPS to work. Section 5 presents our novel library implementation of TPS, exploiting recent features of the Java language. Section 6 evaluates generics and reflection in Java through our library implementation, and discusses the impact of Java's type system on these features. Section 7 discusses various issues, such as alternative languages and/or language features and alternative abstractions for distributed interaction. Section 8 compares our efforts with related notions of typed event-based programming, and programming languages with specific support for distribution. Section 9 presents conclusions drawn from our experience.

2. PRELIMINARIES

This section provides background information on the publish/subscribe paradigm, and introduces the system model and assumptions made. More insight into related event-based models and systems, including a notion of event *type*, is provided in Section 8.

2.1 A Historical Survey of Publish/Subscribe

The publish/subscribe paradigm has found wide acceptance wherever a one-to-many interaction style is required, including many applications in finance and telecommunications. The propagation of *news* (in the sense of news groups) or of any kind of *update* reflects such interaction models.

2.1.1 *It All Began ....* One of the main spiritual ancestors of the publish/subscribe abstraction is the group communication paradigm [Powell 1996], which has found widespread application in fault-tolerant distributed systems as the foundation for *replication* [Birman 1993]. Based on the observation that a set of distributed processes cannot even reach a consensus on a common value if these processes are error-prone (e.g., some of them might fail) [Fischer et al. 1985], many distributed systems are built by making use of the group paradigm as a *communication-centric*, more than a *concurrency-centric*, mechanism (see Section 8.2). This helps in avoiding direct synchronization of remote (failure-prone) parties. The incentive for this consists in that mutual exclusion is even harder to solve (e.g., requires even stronger assumptions to be solvable) among a distributed set of failure-prone processes than consensus [Delporte-Gallet et al. 2005]. Synchronization of parties can be achieved more indirectly by making use of multicast algorithms with specific ordering guarantees that build on minimal assumptions. For example, total order broadcast has been shown to be equivalent to consensus. Group communication, however, also encompasses one-to-many interaction with weaker guarantees.

2.1.2 *Topic-Based Publish/Subscribe.* Most early commercial publish/subscribe systems proposed schemes based on groups. Subscribing to a group "StockQuotes" entails becoming member of this group (e.g., TIB/Rendezvous [TIBCO 1999], SmartSockets [Talarian Corp. 1999], IBus [Altherr et al. 1999]). These groups also often appeared under the name of topics, leading to the widely adopted term *topic-based publish/subscribe*. Most of these systems support one or several specifications out of a proliferating family of standards, for example, Java message service (JMS) [Happner et al. 2002], CORBA event and notification services [OMG 2000, 2001b], or JavaSpaces [Freeman et al. 1999], which promote some form of first-class named communication channel for which the name is typically the topic.

The term subject-based publish/subscribe is sometimes used as a synonym for topic-based publish/subscribe, while sometimes, the *hierarchical disposition* of names is viewed as specific to subject-based publish/subscribe. All prototypes cited previously propose a hierarchical orchestration of groups with a URL kind of notation for referring to these groups (e.g., "/StockQuotes/Telco"), where a subscription to a node in the hierarchy triggers subscriptions to the entire subtree.

2.1.3 *Content-Based Publish/Subscribe.* The *content-based* (also *property-based*) publish/subscribe variant originated in academia (e.g., Siena [Carzaniga et al. 2000], Gryphon [Aguilera et al. 1999]), and has made its way into most of the aforementioned systems and specifications. In content-based

```
public class DynamicEvent {
  public static final int IntegerType = 1;
  public static final int FloatType   = 2;
  public static final int StringType  = 3;
  ...
  public void addInteger(String fieldName, int i) {...}
  public void addFloat(String fieldName, float f) {...}
  public void addString(String fieldName, String s) {...}
  ...
  public int getInteger(String fieldName) throws WrongTypeException {...}
  public float getFloat(String fieldName) throws WrongTypeException {...}
  public String getString(String fieldName) throws WrongTypeException {...}
  ...
  public String[] getFieldNames() {...}
  public int getFieldType(String fieldName) {...}
  ...
}

public class WrongTypeException extends Exception {...}
```

Fig. 1. Dynamically structured events.

publish/subscribe, subscribers can express interest in events with certain runtime properties.

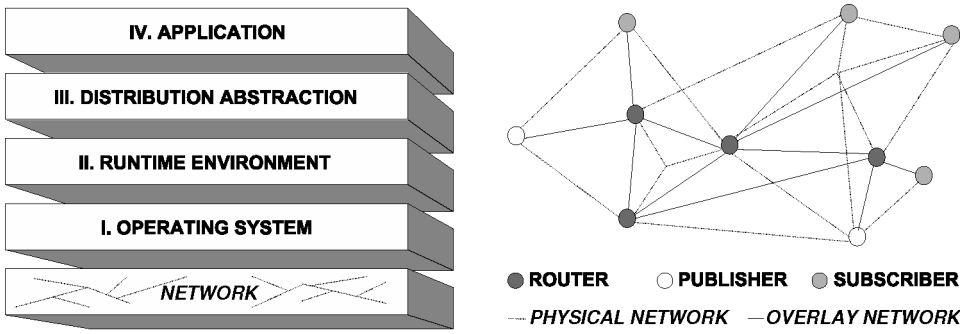
Subscriptions are expressed as predicates based on these properties, and these subscription patterns are viewed as content filters when (generated and) applied by the communication middleware.

In most content-based systems, events are viewed as sets of values of primitive types, or records, and properties of events are viewed as fields of such structures. Dynamically structured events (self-describing messages [Oki et al. 1993]) supporting some form of *introspection* on their content, as illustrated in Figure 1 with Java syntax, can be found in most content-based publish/subscribe systems. As an example, an event representing a stock quote would have a field called `price` of floating point type. Likewise, most standardized APIs (e.g., Happner et al. [2002], OMG [2000, 2001b]) view properties as characteristics explicitly attached to events.

Subscription criteria consist of desired values for given properties, and can be expressed in various ways. The most prominent approach consists in using a subscription language (complementary to the programming language) to express property-value pairs (e.g., Bacon et al. [2000]; OMG [2000]) such as "`'price'<100.0`." Relying on such pairs enables very efficient realizations, since computational overhead is reduced by allowing events to be represented and handled by indexed structures. Alas, such languages provide little safety, since a misspelled property name can only be detected at execution. Examples of query languages are provided in Section 8.1, along with alternative mechanisms for content-based subscription.

2.1.4 *Event Correlation.* Event correlation [Mansouri-Samani and Sloman 1997; Krishnamurthy and Rosenblum 1995] is a spin-off of the publish/subscribe abstraction. With event correlation, subscribers can express interest in being notified upon the occurrence of specific combinations of events only.

Event correlation has been sometimes viewed as a publish/subscribe style in itself. We view event correlation as orthogonal to the main addressing scheme. For example, a subscriber could be interested in being notified of the occurrence



(a) layers (b) nodes

Fig. 2. Architecture overview.

of a pattern of events appearing under respective topics. A straightforward event correlation scheme could hence be built by permitting the logical combination of single content filters.

More advanced event correlation models introduce many additional issues, which represent active research topics, and we thus do not consider this model in the following.

2.2 Reference Architecture

To simplify presentation, as well as comparison with prominent publish/subscribe systems, we assume in the following that a "reference implementation" underlies the publish/subscribe communication substrate. More precisely, we assume a topic-/content-based publish/subscribe engine providing hierarchical topics, together with a SQL-like query language (such as that used in JMS [Happner et al. 2002]).

2.2.1 *Vertical Decomposition.* We assume that the complete system covers the layers depicted in Figure 2(a). Above the actual network itself, these layers are, from bottom to top:

(I) Operating system: The operating systems provide basic primitives for networking and concurrency, such as sockets and threads.

(II) Runtime environment: We assume a language runtime such as the Java virtual machine, which we focus on in major parts of this article. The .NET runtime, which we consider in a second step, is another example. We furthermore assume libraries for accessing and managing rumtime internals and operating system mechanisms such as the aforementioned to be part of this layer.

(III) Distribution abstraction: This layer is assumed to provide higher-level (than earlier mentioned plain sockets) abstractions for distribution, such as remote method invocations or type-based publish/subscribe.

(IV) Application: Distributed applications make use of one or several of these abstractions for distributed programming.

We assume that the distributed applications and distribution abstractions are written in the same programming language, and that the runtime environment does not provide any specific hooks or functionalities for any of these individual abstractions. Rather, we are interested in general support for such abstractions.

2.2.2 *Horizontal Decomposition.* The underlying publish/subscribe engine, which can be viewed as part of Layer III in Figure 2(a), is presumed to construct an overlay network, such as that presented in Figure 2(b). This overlay network exploits information regarding:

—*Content filters.* Content filter semantics are taken into account (to construct the overlay, such as) to perform efficient and scalable filtering. In particular, router nodes in the overlay network can identify and take advantage of redundancies between the content filters of different subscribers [Aguilera et al. 1999].

—*Locality.* The proximity of subscribers (and publishers) in terms of "location" (e.g., communication latency) is usually orthogonal to their proximity in terms of interests, and thus to the amount of overlap in respective content filters [Eugster and Guerraoui 2002]. Combining both notions of proximity yields the best results.

As briefly alluded to in Section 1, routing and filtering events does not necessarily require the deserialization of these events at each filtering step. A publish/subscribe engine can transform dynamic events to an even lower-level representation (e.g., based on XML [Baehni et al. 2003]), and make use of highly efficient dedicated routers (see Figure 2(b)) involving a different stack of layers than those of Figure 2(a). For the following, it is only important to keep the rationale behind design choices of content-based publish/subscribe systems in mind, not the actual realization of the corresponding mechanisms.

3. TYPE-BASED PUBLISH/SUBSCRIBE

In this section, we first describe the type-based publish/subscribe (TPS) paradigm from a general point of view, in other words, its principles and concepts without deployment within a particular statically typed language in mind, and then discuss incarnations of TPS in a programming language such as Java.

3.1 Principles

Together with industrial partners (see Acknowledgements), we have come up with a set of requirements that a typed publish/subscribe abstraction should provide:

—*Type safety* (*TS*): In a statically typed language, type safety is enforced in local interactions, and as far as possible, should also be provided for remote object interaction. Type errors should be recognized at compilation, alleviating the already cumbersome debugging of distributed applications.

—*Encapsulation preservation* (*EP*): Events are to be considered as objects, and hence as instances of (abstract) types. Their implementation details should not be systematically revealed, particularly not in the goal of describing subscription criteria.
—*Application-defined events* (*AE*): Events (event types) should be defined as part of the application design, with minimal imposed design choices.
—*Open content filters* (*OF*): To support efficient dissemination of events, content filters should be expressed in a manner which provides the underlying publish/subscribe engine insight into these filters for the configuration of routing and filtering algorithms.
—*Event semantics* (*ES*): To express some form of *Qualities of Service* (QoS), different basic semantics should be assignable to events.

Achieving these principles in combination is not straightforward, as already implied previously. For instance, providing content-based subscription to events by preserving encapsulation of these event objects (**EP**, that is, not systematically expressing subscriptions as property-value pairs, ruling out any query languages) is already commonly pictured as a contradiction *per se* [Koenig 1999]. Further complexity is added by requiring transparency of content filters (**OF**, that is, giving the publish/subscribe implementation—see Layer III in Section 2.2.1—full access to subscription criteria in order to optimize the filtering of events).

Satisfying the preceding principles in the context of subscriptions boils down to providing a means of expressing these subscriptions in the considered programming language itself. This constitutes the main challenge of TPS, which distinguishes it from related notions of typed publish/subscribe, as presented in Section 8.1. Roughly, the task of the TPS abstraction can be imagined as consisting of transforming events to a lower-level representation, such as dynamic events, and constructing queries (see Section 2.1.3) from content filters expressed in the programming language. We will not attempt to *quantify* expressiveness at the language level, nor its impact on scalability or efficiency; any sensible scheme should, however, retain at least the expressiveness and efficiency of a "classic" approach based on dynamic events and a query language, as discussed in Section 2.2.

3.2 Model

The core idea underlying our integration of publish/subscribe with objects consists in viewing events as first-class citizens.

3.2.1 *Events.* By considering events as first-class citizens, that is, not as specific constructs (e.g., Haahr et al. [2000]), but as specific application-defined objects, we strongly enforce **TS** and **AE** (see Thomas [2004]). To emphasize the object nature of events, we call these *event objects*, or to abbreviate notation, simply *obvents*.

Similarly to Oki et al. [1993], we distinguish between two main categories of objects, but also introduce two further (sub)types:

—*Unbound objects.* Unbound objects are *locality-unbound*, that is, their semantics do not depend on any local resource. Such objects can be transferred to another address space (in Oki et al. [1993], these are termed *data objects*).
  —*Obvents.* Obvents represent a specific kind of unbound object. Such objects are used to notify events in the context of publish/subscribe interaction (and can, in a nested way, contain other unbound objects).
—*Bound objects.* These objects are local to an address space (locality-bound) and remain in this address space during their entire lifetime. They may make use of specific local resources (*service objects* in Oki et al. [1993]).
  —*Subscribers.* Potentially, any bound object could take the role of subscriber, but in general, only particular objects subscribe to obvents. Of course, subscribers could be mobile; presentation is, however, simplified by leaving aside this possibility.

Note that a generic notion of events can also capture dynamic events, as these can still be used whenever the late binding they offer is really required. This retains flexibility.

3.2.2 *Publishing Obvents.* The only contracts between publishers and subscribers are the types of published obvents. A publisher has no explicit notion of "destination" when publishing such an obvent. The set of destinations is implicitly and dynamically defined by the subscribers whose criteria match that obvent.

A published obvent o acts as a template, and a publication can be pictured as a distributed form of object cloning where a clone of the prototypical object o is created for every subscriber. The set of processes where this action will take place is given by the set of processes that are willing to host such objects, in other words, whose subscription criteria match the template object. Inversely, a subscription expresses the desire of obtaining a clone of every published object which corresponds to the subscription criteria.

More precisely, a distinct copy of a published obvent is created for each subscriber:

—*Obvent global uniqueness.* Suppose that an obvent $o1$ is published from an address space $a1$: if an address space $a2$ contains two subscribers $s1$ and $s2$, these will receive references to two new distinct clones of $o1$, say $o2$ and $o3$.
—*Obvent local uniqueness.* In the previous scenario, if the address space $a1$ also contains a subscriber $s3$, then $s3$ will receive a reference to a new obvent $o4$.

The notion of cloning here corresponds to a *deep* cloning: When a clone of an object is created, its fields are recursively cloned [Gregono and Sakkinen 2000].

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

3.2.3 *Type-Based Subscription.* The main subscription criterion for consumers in TPS becomes the (abstract) type of the event object of interest. When subscribing to a type T, we express interest in instances of T, that is, instances of any types which *conform to* T.

A subscription can in this sense be seen as a contract for hosting objects that are 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.

By using the types of obvents as the basic subscription criterion, we strongly leverage type safety (**TS**): By matching the notion of event *kind* with that of event *type*, that is, using the type scheme of the programming language as the subscription scheme, the type of "received" event is known, and compile time type-checks can be performed on corresponding formal arguments.

Yet, 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.

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. Certain restrictions on the semantics of content filters can, however, help to ensure an efficient and scalable implementation of the underlying TPS engine, as mentioned already.

3.3 Type-Based Publish/Subscribe in Java

The semantics of TPS hence strongly depend on the interpretation of conformance in the considered context, which itself 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 also be based on a neutral *event definition language* (EDL) to enforce interoperability, leading to a second-class TPS package [Baehni et al. 2003]. Note in this context that interoperability is also an argument often used for promoting query languages. The validity of this argument is, however, somewhat limited by the existence of a plethora of different query languages (see Section 8.1).

In the following, we illustrate TPS through the Java programming language.

3.3.1 *Background: Java Types.* In many strongly typed object-oriented languages like C++ [Ellis and Stroustrup 1992] or Eiffel [Meyer 1992b], 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 quasi-identical.

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 defining an interface which can subtype several superinterfaces: an interface I1 which

```
package java.tps;

import java.io.*;
import java.rmi.*;

/* obvents */

public interface Obvent extends Serializable {...}
public interface ReliableObvent extends Obvent {}
public interface CertifiedObvent extends ReliableObvent {}
public interface TotalOrderObvent extends ReliableObvent {}
public interface FIFOOrderObvent extends ReliableObvent {}
public interface CausalOrderObvent extends FIFOOrder {}
public interface TimelyObvent extends Obvent {
  public long getTimeToLive();
  public long getBirth();
}
public interface PrioritaryObvent extends Obvent {
  public int getPriority();
}

/* exceptions */

public abstract class NotificationException extends RemoteException {...}
public final class CannotPublishException
    extends NotificationException {...}
public final class CannotSubscribeException extends ...
public final class CannotUnsubscribeException extends ...
```

Fig. 3. Basic obvent types and exceptions for TPS in Java.

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's type.

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.

3.3.2 *Subscriptions and Java Types.* As a consequence of the intertwining of types and classes in Java, it makes sense to support subscriptions to interfaces as well as to classes. We could, however, argue for supporting interfaces only, both for alignment with Java RMI (in which remotely invocable types must be interface types), and for promoting encapsulation (interfaces can not declare fields). In the following, we will nonetheless support subscriptions to classes as well, for illustration purposes.

3.3.3 *Obvents in Java.* 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`.

The basic Java Obvent type (Figure 3) thus subtypes former type. This eases the implementation of our obvent model in general. Such a built-in (default) serialization mechanism strongly supports **AE** by relieving developers from the burden of implementing specific operations or hooks in their obvents. This way,

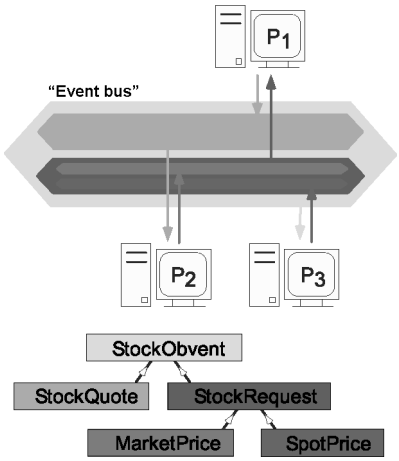


Fig. 4. Stock trading with type-based publish/subscribe.

the design phase of obvents can be cut down to the essential meaning of the represented logical event. With a simple convention, for example, event field `x` is accessible (only) through a method `getX()`, events can easily be transformed to a lower-level representation of the underlying publish/subscribe reference engine.

In our model, such characteristics are associated with obvents, and should thus be part of these obvents. Indeed, it makes sense for every obvent to reflect 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 types are the only (implicit) contracts between publishers and subscribers, we have chosen to use these to express a simple form of QoS mandated by **ES**. A per obvent description of QoS would of course offer more flexibility, and might be more advisable in certain contexts [Araujo and Rodrigues 2002], but would come at the price of requiring negotiation. Figure 3 depicts Java types corresponding to different semantics. For example, `ReliableObvent` is the root type of all obvents that are to be conveyed with end-to-end reliability guarantees. Each distinct obvent type can lead to the use of an individual protocol by the underlying engine.

3.4 Running Example: Stock Trade

Figure 4 illustrates the intuitive idea underlying TPS through a recurring example for publish/subscribe interaction, which is stock trading. 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 either only at the end of the day or once another given criterion is fulfilled. As outlined in Figure 4, these different kinds of events result in corresponding custom obvent types, rooted at the `StockObvent` type depicted in Figure 5 (details of these types are omitted here for simplicity).

```
import java.tps.*;

public class StockObvent implements ReliableObvent {
  private String company;
  private int amount;
  private float price;
  public StockQuote(String company, int amount, float price) {
    this.company = company;
    this.amount = amount;
    this.price = price;
  }
  public String getCompany() { return company; }
  public int getAmount() { return amount; }
  public float getPrice() { return price;}
  ...
}

public class StockQuote extends StockObvent {
  private long time;
  public StockQuote(String company, int amout, float price, long time) {
    this(company, amount, price);
    this.time = time;
  }
  public StockQuote(String company, int amount, float price) {
    this(company, amount, price, System.currentTimeMillis());
  }
}
```

Fig. 5. Stock quotes in Java.

Market price requests can expire, and for the broker's (e.g., p2) convenience, an intermediate party (p3), for example, a bank, might also handle such requests on his or her behalf, for instance by issuing spot price requests to the stock market once the broker's criteria are satisfied. In the following, we will assume that p2 in Figure 4 is only interested in stock quotes from a company called "Telco" that cost less than $100. In Java, a corresponding content filter expressed through a formal argument quote could look like the following:

```
quote.getCompany().equals("Telco") && quote.getPrice() < 100.0
```

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

These concepts are made concrete in the next two sections.

4. JAVA$_{PS}$

In this section we present a first approach to implementing TPS by extending the Java language. This approach illustrates inherent difficulties of implementing TPS.

4.1 Syntax

More precisely, this approach consists in augmenting the Java language with two specific primitives, for publishing obvents and subscribing to obvent types, respectively [Eugster et al. 2001]. In doing so, Layer III of Figure 2(a) permeates Layer IV.

4.1.1 *The* publish *Primitive.* With this primitive, an obvent can be published, which implies that it is asynchronously sent to any concerned subscriber.

SubscriptionExpression:
subscribe *SubscriptionDeclaration*
SubscriptionDeclaration:
SubscriptionDeclarator FilterBody HandlerBody
SubscriptionDeclarator:
(*SubscriptionFormalParameter*)
SubscriptionFormalParameter:
ObventType Identifier
FilterBody:
Block
HandlerBody:
Block
ObventType:
InterfaceType

Fig. 6. Detailed syntax of subscription statements.

Following the Java language specification grammar [Gosling et al. 2000], based on a LALR(1) syntax, we introduce a new statement (our specific definitions are typed in **bold**).

PublishStatement:
publish *Expression* ;

Here, *Expression* is a non-null expression of type `Obvent`, as opposed to most libraries relying on Java serialization: In Java, a serializable root type is often faked by using formal arguments of the root type `java.lang.Object`, yet expecting an actual argument to be of the more specific type `java.io.Serializable` and throwing an exception if this expectation is not fulfilled. We prefer detecting such type errors at compilation. The `publish` primitive can, however, throw an exception of type `CannotPublishException` (see Figure 3) to signal communication problems in the underlying TPS engine.

4.1.2 *The* `subscribe` *Primitive.* We introduce a second primitive, `subscribe`, to express subscriptions. A subscription expression has the following syntax in Java (details are given in Figure 6):

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 *InterfaceType* (Sect. 4.3 in Gosling et al. [2000]). The filter represented by the first block must return an expression of type boolean, while the evaluation block, which we refer to as the *obvent handler* or simply *handler*, returns nothing. An exception `CannotSubscribeException` is thrown if any problems appear when advertising the subscription. A successful subscription returns an object of type `Subscription` (see Figure 7). Such a handle uniquely identifies a subscription, and is required for the activation and deactivation of subscriptions.

```
package java.tps;

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

Fig. 7. Subscription handles in Java.

4.2 A Closer Look at Subscriptions

In short, a subscription expression combines the subscription to a type T with: (1) a closure declaration representing a content filter of the form

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

and (2) the declaration of a second closure of the following form:

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

4.2.1 *Background: Closures.* Different notions of "local" closures have appeared in the literature, such as the *block closure* in Smalltalk [Goldberg and Robson 1983], or the *anonymous function* in Cecil [Chambers 1995]. The different notions of closures vary mainly in the degree of self-containment they advocate. A first-class block closure in Smalltalk can use any variable in scope of the closure declaration (at compilation), and these variables are bound for the entire lifetime of the closure, even if the closure is then executed in a context where some of these variables are not visible. To avoid this binding of variables, only final variables in scope at compilation can be used within an *anonymous class* in Java.

4.2.2 *Obvent Handlers.* Obvent handlers, or simply handlers, adopt the latter closure semantics, as they represent an intuitive way of handling callbacks from the underlying event dissemination system and are executed locally. The use of such closures enables the regrouping of all code related to individual subscriptions in succinct expressions.

By viewing these closures as objects, the handlers take on the role of the subscribers, outlined in Section 3.2.1.

4.2.3 *Content Filters.* Akin to handlers, content filters are closures with a specific signature. Besides the concentration of subscription-related code, the use of such a syntax in the case of filters is further motivated by the desire not only to confine the code for filtering, but to "reveal" it to the TPS implementation. This enables the configuration of an overlay network to be responsible for routing obvents by: (1) constructing queries from filters and their application on foreign hosts, as well as (2) factoring out redundancies between these filters from different subscribers.

The compilation of these filters is hence deferred, similar in a sense to the deferred code evalution underlying *two-level programming* [Nielson and Nielson 1988] (or generalized to more than two levels: *multistage programming*, as advocated by MetaML [Taha and Sheard 1997]).

4.2.4 *Restrictions on Content Filters.* Java$_{PS}$ supports the same semantics for closures representing content filters as for anonymous classes. However, since the goal is not to quantify the impact of expressiveness on the scalability of the underlying publish/subscribe engine, we restrict the semantics of these closures to make the inference of remotely evaluatable queries from content filters more tractable:

—*Methods.* The only method invocations allowed in a content filter are: (nested) invocations on its variables.
—*Variables.* The only variables allowed in a content filter are: (1) the formal argument representing a filtered obvent, (2) local variables, and (3) final outer variables. The latter two types of variable are restricted to primitive types (e.g., float) and their object counterparts (e.g., `java.lang.Float`), as well as `java.lang.String`.
—*Operators.* Any kind of comparison operators or Boolean operators are allowed.
—*Control structures.* Only `if` / `else` control structures are supported.

With these restrictions, content filters become reducible to a set of elementary predicates combined by Boolean operators (`if` / `else` can be transformed to logical ANDs and ORs).

4.3 Illustration

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

```
StockQuote quote = new StockQuote("Telco", 100, 90.0);
publish quote;
```

Subscribing to stock quotes can be expressed as follows:

```
Subscription quoteSubscription = subscribe (StockQuote quote)
{
   return (quote.getCompany().equals("Telco") &&
           quote.getPrice() < 100.0);
}
{
   System.out.println("Got offer: " + quote.getPrice());
};
quoteSubscription.activate();
```

Note that the content filter, that is, the first block, is expressed in Java$_{PS}$ with the exact same code as in Section 3.4.

4.4 Implementation

Java$_{PS}$ has been implemented with a specific compiler `psc`. More precisely, we have made use of the extensible compiler JaCo [Zenger and Odersky 2001] to deal with our specific primitives. We refer to our instance of this compiler as `psc`.

```
import java.tps.*;

public final class TAdapter {
  public static Subscription subscribe(LocalFilter l, RemoteFilter r,
                                       Subscriber s) {...}
  /* if T is a class */
  public static void publish(T t) throws CannotPublishException {...}
  ...
}
```

Fig. 8. Adapter for a type T.

4.4.1 *Typed Adapters.* To avoid making the Java virtual machine distribution-aware, and to exploit a class-based dissemination implemented by our engine and its multicast algorithms (mapping obvent classes to groups [Eugster and Guerraoui 2002; Baehni et al. 2004]), we adopt the notion of *adapter* from Oki et al. [1993]. Adapters are intermediate entities between the communication substrate and application, whose role in TPS consists mainly in mediating between serialized data and typed obvents. Our adapters are type-specific, and are generated for each obvent type by the psc compiler. For any given obvent class C, psc generates a class CAdapter with code for publishing instances of C, and subscribing to C. To support subscriptions to abstract types (interfaces), psc generates a class IAdapter for any abstract obvent type I with code for subscribing to instances of I.

Correspondingly, psc transforms publish statements and subscribe expressions to invocations of methods in corresponding adapter classes. Figure 8 depicts an adapter class for a given obvent type T.

4.4.2 *Publishing.* Since a published obvent is disseminated through the adapter for its dynamic type, which is only known at runtime, a *PublishStatement* cannot be directly transformed to a call to publish on the corresponding adapter class. Hence, a publish() method is added to the Obvent interface in Java (Figure 9), 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:

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

provided, of course, that o's static type can be widened to Obvent (see Section 4.1.1). The publish() method then can generate a dynamic event from the obvent passed as a parameter. To support nested queries involving obvent fields, such as the first predicate of our running example, dynamic events can be generated as "flattened" representations of the corresponding obvents.

```
package java.tps;

import java.io.*;

/* obvents */

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

/* top level */

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

/* filters */

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

/* handlers */

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

Fig. 9. Details on Obvent, Subscription, and further types in Java$_{PS}$.

4.4.3 *Subscriptions.* By similarly transforming subscriptions to (static) calls to corresponding obvent types, subscriptions 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 the subscribe() method in class TAdapter.

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

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

This is an instance of an anonymous class implementing the Subscriber type given in Figure 7:

```
new Subscriber() {
  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.

4.4.4 *Content Filters.* The handling of filters represents the most complex task during precompilation. An anonymous class (here representing a unary

predicate of type LocalFilter, depicted in Figure 9) is generated for every content filter. It represents the full filter, and is applied locally. More precisely, a subscription such as:

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

is transformed into an invocation of the corresponding adapter class:

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

We note that psc also generates an intermediate representation of the filter, which can be used to generate query expressions that are passed to the underlying publish/subscribe engine. These RemoteFilters thus resemble *application-specific handlers* [Engler et al. 1996], low-level message filters, except that the latter kind of filters is applied locally and expressed in a neutral specification language, while our filters more resemble parse trees by promoting the use of the native language syntax.

4.4.5 *Representing Subscriptions.* Our precompiler generates two tree-like constructs, which are more specific than for instance the parse trees used in Smalltalk [Rivard 1996] due to the restrictions imposed (see Section 4.2.4).

—*Invocation tree.* First, a representation of the invocations made in the filter is generated. The root represents the filtered obvent, and every edge leading to a nonleaf node represents a method invocation. A nonleaf node represents the value obtained by applying the methods of edges on the path down to this node in a nested fashion to an instance of the considered obvent type. Edges leading to leaf nodes are comparisons or methods yielding a Boolean value. A leaf node thus stands for the outcome of a condition on obvents of the considered type. Of course, edges can also represent field accesses.

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

For the following, we assume that the TPS engine applies (nested) methods on events, including arguments locally at the subscriber sites, and generates queries from direct (nested) field reads combined with comparisons for remote application by the underlying topic-/content-based engine, as described in Section 2.2.

5. TPS AS A LIBRARY

The experience gathered with Java$_{PS}$ has motivated and served us to implement a library version of TPS, roughly consisting of promoting the adapter concept introduced in the previous section to first class. This work, which has enabled the exploration of recent and (at that point) future concepts of Java, has also benefitted from previous efforts on using *collections* (*distributed asynchronous collections*, or DACs [Eugster et al. 2000; Eugster and Guerraoui 2001]) as such adapters.

5.1 Generics

With a library approach, that is, by making use of only standard class libraries, we cannot rely on the generation of type-specific adapters. Yet, the handler and content filter of a subscription have to "agree" on the obvent type subscribed to, which can be different for every subscription. As the choice of type (and agreement on it between handler and content filter) is sealed by instantiating the TPS abstraction, an approach that immediately comes to mind consists of making the adapters of Java$_{PS}$ *generic* [Milner 1977].

5.1.1 *Background: Generics in Java.* While languages like C++ or Ada 95 incorporate generic types, languages such as Java or Oberon were initially designed to replace variable types by the root of the type hierarchy.

Several dialects of the Java language, providing some form of generics, have been proposed. *Generic Java* (GJ) [Bracha et al. 1998], an approach supporting *parametric polymorphism* (*F-bounded polymorphism* [Canning et al. 1989]) through a specific compiler, has quickly taken the lead. GJ has served as base for Sun's own efforts to integrate generics into Java, which became concrete in Java 1.5 [Bracha 2004].

5.1.2 *Generic Adapters with GJ.* We implemented the first generic adapters for TPS based on GJ. These provide type safety, that is, they supercede explicit casts in TPS applications, without requiring the generation of type-specific code. The resulting generic `Adapter` type is depicted in Figure 10. The `Subscription` type, as well as `Subscriber` type (used behind the scenes in Java$_{PS}$), have accordingly been added respective type parameters.

5.2 Reflection

The big open question now is how to support the type safe expression of content filters in a way that provides third-party publish/subscribe libraries with an insight into these filters. Reflection seems to be the ideal means of providing static type-checking while deferring the "interpretation" of the corresponding code.

5.2.1 *Background: Behavioral Reflection in Java.* A candidate mechanism for the type safe expression of content filters was anticipated with the integration of a limited mechanism for behavioral reflection in Java 1.3 through the `Proxy` class in package `java.lang.reflect`. Invoking its `getProxyClass()` method (see Figure 11) leads to creating (unless this already happened) a proxy

```
package java.tps;

public final class Adapter<T> {
  public void publish(T obvent) throws CannotPublishException {...}
  public Subscription<T> subscribe(Subscriber<T> subscriber) {...}
  ...
}

public final class Subscription<T> {
  public void activate() throws CannotSubscribeException {...}
  public void deactivate() throws CannotUnsubscribeException {...}
  ...
  public void setSingleThreading() {...}
  public void setMultiThreading(int maxNb) {...}
  ...
  public T constrain() {...}
}

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

Fig. 10. Basic interfaces in the library implementation.

```
package java.lang.reflect;

public class Proxy implements Serializable {
  public static Class getProxyClass(ClassLoader l, Class[] is)
    throws IllegalArgumentException {...}
  public static Object newProxyInstance(ClassLoader l, Class[] is,
                                        InvocationHandler h)
    throws IllegalArgumentException {...}
  public static InvocationHandler getInvocationHandler(Object p)
    throws IllegalArgumentException {...}
  ...
}

public interface InvocationHandler {
  public Object invoke(Object p, Method m, Object[] as) throws Throwable;
}
```

Fig. 11. `Proxy` and `InvocationHandler` types.

class as a subclass of `Proxy` which implements a set of interfaces specified by `Class` metaobjects. Proxy classes are created directly as byte code, and automatically loaded and linked. The `newProxyInstance()` method also instantiates the (possibly generated) proxy class. Such a dynamic proxy object can then be used in a consistent manner wherever an expression of (one of) the type(s) it was created for is expected.

While the introspection mechanisms provided in Java since version 1.1 enable the reification and dynamic invocation of methods (deferring to runtime the choice of *which* method to invoke), dynamic proxies allow the interception of (in addition) statically typed method invocations, and the performing of any action within the confines of these invocations (deferring to runtime *what to do upon* invocation of a method).

In the following illustrations, dynamic proxies (as well as generics) are idealized. Caveats are the subject of Section 6.

5.2.2 *Recording Predicates with Dynamic Proxies.* We devised a scheme allowing the developer to express content filters by making use of proxies as

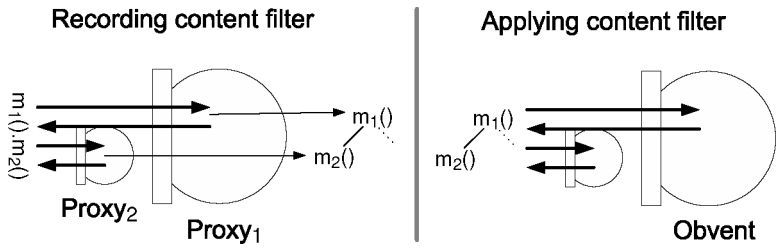


Fig. 12. Expressing content filters with dynamic proxies.

formal arguments for expressing corresponding invocations. The proxy then records these invocations (reifying these) such that they can be replayed on the effective obvents for filtering (by simply performing them, or "applying" them differently, e.g., by reading the corresponding field in the case of a field access method). This is illustrated by Figure 12, where one can imagine that the `getCompany()` method is invoked on a first proxy. After recording the invocation (along with arguments, if any), a second proxy, mimicking the return value, is returned. This second proxy is then recursively invoked, in this context through the `equals()` method.

To support this scheme, the `Subscription` class, used together with first-class adapters, has been added a method `constrain()`, which returns a dynamic proxy of the exact type of the subscribed type. An identical proxy can be used to record several predicates which are conjoined. A disjunction of predicates can be achieved by expressing these predicates through different proxies obtained through successive calls to `constrain()`.

5.3 Illustration

The use of these generic and reflective first class adapters is best illustrated by the following example of stock quote publication:

```
Adapter<StockQuote> quoteAdapter = new Adapter<StockQuote>();
StockQuote quote = new StockQuote("TelcoOperators", 100, 90.0);
quoteAdapter.publish(quote);
```

In the following, we attempt to express the criterion of the ongoing example on stock quotes, that is, interest in all stock quotes from the "Telco" group. The subscriber is created through an anonymous class:

```
Adapter<StockQuote> quoteAdapter = new Adapter<StockQuote>();
Subscription<StockQuote> quoteSubscription =
  quoteAdapter.subscribe(new Subscriber<StockQuote>() {
    public void notify(StockQuote quote) {
     System.out.println("Got offer: " + quote.getPrice());
    }
  });
Stockquote formalQuote = quoteSubscription.constrain();
formalQuote.getCompany().equals("Telco");
formalQuote.getPrice() < 100.0;
quoteSubscription.activate();
```

5.4 Implementation

Figure 13 presents the skeleton of the implementation of the generic Subscription class, which is instantiated upon invocation of the subscribe() method in class Adapter.

5.4.1 *Background: Dynamic Proxies and Invocation Handlers.* An invocation performed on a dynamic proxy object is reified and passed to an InvocationHandler (Figure 11) object, associated to this proxy at instantiation, through the handler's invoke() method. The arguments for invoke() include: (1) the proxy object on which the method was originally invoked, (2) a meta-object representing the method (Method) that was originally invoked, and (3) the effective arguments (an array of Objects) to this invocation. This makes the invoke() method capable of handling *any* method invocation. Arguments of primitive types are transformed to the corresponding wrapper types.

5.4.2 *Invocation Chains.* Accordingly, the implementation of the Subscription class introduces two inner classes, called InvocationRegistrar and Invocation, to register invocation chains with dynamic proxies and to represent these chains, respectively: Whereas the former class is an implementation of the InvocationHandler interface and is used to "interpret" each invocation recorded, the latter is used subsequently to store the information about the method that was actually invoked, along with any arguments.

The constrain() method of the Subscription class thus creates an instance of class Invocation, which in turn creates and returns a proxy which is associated an instance of InvocationRegistrar. The latter object does the actual job of recording the invocation, which it passes back to its associated Invocation object. In the case where the invocation involves a return object of a type other than Boolean, the handler recursively creates a new Invocation, etc.

An instance of Subscription stores all invocation chains recorded through it in a vector, called predicates, in the code sketched in Figure 13. As outlined previously, these chains are then conjoined.

It is important to note that, as mentioned previously, both reflection (dynamic proxies) and generics are idealized. Hence, the preceding example can not be made to work exactly as such, due to restrictions of these features. These limitations are discussed in the next section.

6. EVALUATION

In this section, we present an evaluation of the previously presented library implementation, comparing it with Java$_{PS}$. We point out weaknesses of current implementations of generics and reflection, as well as of other features, in the Java programming language [Damm et al. 2004].

6.1 Generics

Generics have proven extremely useful in TPS for implementing type safe interfaces. However, for presentation simplicity, the example in the previous section passed over an important limitation.

```
package java.tps;

public final class Subscription<T> {
  private Class type;
  private Vector predicates;
  ...
  Subscription(Class type) { this.type = type; }
  ...
  /* return a ''fake'' instance of T */
  public T constrain() {
    Invocation nestedInvocation = new Invocation(type);
    predicates.add(nestedInvocation);
    return (T)nestedInvocation.getProxy();
  }
  ...

  private static class Invocation {
    private Class type;
    private Invocation nested;
    private Method method;
    private Object[] args;
    ...
    private Invocation(Class type) { this.type = type; }
    ...
    private void setNested(Invocation nested) { this.nested = nested; }
    private void recordInvocation(Method method, Object[] args)
      { this.method = method; this.args = args; }
    private Object getProxy() {
      return Proxy.newProxyInstance(type.getClassLoader(),
                                    new Class[] { type },
                                    new InvocationRegistrar(this));
    }
  }

  private static class InvocationRegistrar implements InvocationHandler {
    private Invocation invocation;
    private InvocationRegistrar(Invocation invocation)
      { this.invocation = invocation; }

    public Object invoke(Object proxy, Method method, Object[] args) {
      /* register the invocation */
      invocation.recordInvocation(method, args);
      ...
      /* if the method returns a non-bool object, return another proxy */
      Invocation nested = new Invocation(method.getReturnType());
      invocation.setNext(nested);
      return nested.getProxy();
    }
  }
}
```

Fig. 13. Using dynamic proxies for content filter expression.

6.1.1 *Runtime Support.* The implementation presented in the previous section suffers from a lack of runtime support for type parameters in the current implementation of generics in Java. Clearly, although a type parameterized adapter is instantiated for a given type `StockQuote` in the example in Section 5.3, the constructor would require an explicit argument representing a reification of this type precisely such that a given adapter instance would have the possibility of knowing the value of its very type parameter. This is somewhat visible through the implementation outline of the `Subscription` class in Figure 13, which requires the reification of the type parameter (stored in its `type` field) for instantiating dynamic proxies in the

context of invocations of the constrain() method. The implementation of the Adapter class would hence have to be extended somehow, as portrayed next:

```
public class Adapter<T> {
  private Class type;
  ...
  public Adapter(Class type) { this.type = type; ... }
  ...
  public Subscription<T> subscribe(Subscriber<T> s) {
   ...
   return new Subscription<T>(type);
  }
}
```

and instantiated as follows:

```
Adapter<StockQuote> quoteAdapter =
  new Adapter<StockQuote>(StockQuote.class);
...
```

With runtime support, this somewhat redundant (and in fact, error-prone) argument to the constructor could be avoided. In Java$_{PS}$, adapters can obviously be generated by psc with a private field, which is immediately assigned the value of this argument.

Note that in the context of an invocation of the publish() method, the adapter could obtain the type reification of the published obvent by querying the getClass() method on that obvent. However, this reification does not necessarily represent the (precise) value of the type parameter T, but possibly only a subtype thereof. This does not represent a mismatch, but simply makes it impossible to pre-establish a "connection" from such a publisher to a component acting as pure subscriber of T, before any instances have actually been issued by this publisher. In other words, the TPS engine can not preconfigure the multicast algorithm and overlay network employed underneath by setting up topics.

6.1.2 *Beyond TPS—Statically Type Safe Components.* The benefits of generics—roughly speaking, the ensuring of type safety in interactions between library classes (of programming abstractions) and instances of (new) application-defined types—have already become apparent in centralized contexts (e.g., collections). As illustrated by Java$_{PS}$, integrating primitives into a programming language can obviously leverage type safety, as well. However, such an approach provides no flexibility, and tends to pollute programming languages.

In a distributed context, generics become even more important. The increased potential size of applications, along with constraints such as 24×7 requirements, make it even more unfeasible to rewrite, regenerate, or recompile code in order to ensure type safety. Generics, as a mechanism enforcing reuse of code, can fully develop its power in a distributed setting where late binding is required.

Generics are hence not only useful in the context of TPS. A generic lookup service ("registry") could also improve type safety in RMI implementations, and help in avoiding type checks and casts. This is particularly valid in Java RMI, since all remote interactions are statically typed (there is no such thing as a dynamic remote invocation à la *dynamic invocation interface* in CORBA [OMG 2002]), and thus, the type of an obtained reference to a remote object has to be known upfront (see Section 7.2).

No matter what the types involved in remote interaction represent, components making use of the same types must have a means to "connect." This calls for *runtime type information*, for example, the possibility of reifying types. Indeed, verifying how types are related and performing runtime type inclusion checks on objects ensures type safety at the communication infrastructure level. The (current) implementation of generics in Java, which relies on a *homogenous translation* consisting of erasing type parameters at compilation and inserting type casts where necessary, is slim, yet loses track of the values of type parameters.

6.2 Reflection

The latter kind of mechanism is sometimes also viewed as being part of introspection, or more generally, structural reflection. The mechanism for behavioral reflection used for content filter expression in our TPS library prototype has been idealized in the previous section.

6.2.1 *Uniform Proxies.* As a mechanism for general behavioral reflection, proxies have well-known disadvantages. These are mainly caused by the fact that an object, which is associated to a behavior through a proxy by "shielding" it with this proxy, has its own identity distinct from that of the proxy. The resulting drawbacks are well-enumerated and documented in Liebermann [1986], and hence not repeated here.

An important, more specific, shortcoming of the implementation of dynamic proxies in Java is that we cannot create such a proxy (class) for a class [Eugster 2006]. More precisely, a dynamic proxy can only be assigned to a variable of interface type. Hence, the example in the previous section cannot be made to work as such, since the `StockQuote` type is a class (see Section 3.3.2). In order to be able to create a dynamic proxy through the `constrain()` method, the `StockQuote` type would have to be defined as an interface with a corresponding implementation class, as in the following:

```
public interface StockEvent extends ReliableObvent {...}
public class StockEventImpl implements StockEvent {...}
public interface StockQuote extends StockEvent {...}
public class StockQuoteImpl
    extends StockEvent implements Stockquote {...}
```

This could be enforced by mandating the types of obvents used with TPS, that is, obvent types used as values for the type parameter of the `Adapter` class, to be such interface types (see Section 3.3.1). This constraint would enforce

encapsulation by making it impossible to specify content filters based on obvent fields. However, such a constraint cannot be enforced at compilation, and, as shown in the previous application to stock quotes, can become burdensome. But even more importantly, this workaround is, in fact, still not sufficient to express the content filter as presented in Section 5.3. As a nested invocation leads to a recursive creation of dynamic proxies, each return type of a method of type `StockQuote` would have to be an interface type, as well. The same would apply recursively to any return types of methods of these interface types, etc. In the example, this would require the definition of (own) interface types corresponding to the predefined primitive classes, such as for the `String` class (as it is invoked through the `equals()` method in the example). Ultimately, to avoid unexpected restrictions later on, this leads to programming with only interface types for variable and parameter declarations, and making use of classes solely when creating new object instances and assigning these to such placeholders.

6.2.2 *Beyond TPS—Safe Dynamic Composition.* As discussed in Section 5.1, generics enable the implementation of libraries, for example, (representing abstractions) for concurrent or distributed programming in a way that ensures type safe interaction with these libraries. In a centralized setting, certain implementations of generics may suffice to ensure type safety statically, that is, resulting code, once verified at compilation, can be run safely without runtime type-checks.

In a distributed context, however, it is very unlikely that no runtime type-checks have to be performed. Indeed, network channels and, thus, layers "close" to the network and corresponding lower-level libraries are inherently untyped. It appears natural that any implementation of a library for distributed interaction requires explicit type inclusion checks performed dynamically (e.g., against types unknown at compilation). In fact, this requirement holds even in an approach such as Java$_{PS}$, as long as remote interaction is involved.

Furthermore, through the increasing presence of distribution, and hence the omnipresence of distribution-related issues, more recent abstractions for distributed interaction, such as TPS or mobile agents, lead to an intertwining of code that is related to both the application logic, and the distribution.

Reflection can address these needs, by providing means to program in a dynamic style. Reification of both types (*structural reflection*), and computation (behavioral reflection), though potentially associated with performance penalties, appears to be the best way to combine: (1) statically safe interaction between individual components and the abstractions for distributed interaction they rely upon, and (2) the dynamically established connections and compositions of these components.

Similarly, *dynamic class loading* and *serialization* can be viewed as faces of reflection, and have been exploited in both Java$_{PS}$ and our library implementation of TPS as fundamental building blocks for implementing remote interaction.

6.3 Types

Just as the first condition expressed in the content filter of our ongoing example cannot be put to work, as presented in the previous section, the second condition fails by involving an instance of a primitive type.

6.3.1 *Pure Object Type System.* Indeed, when using dynamic proxies for content filter expression, methods returning primitive types cannot be used (except `equals()`, and this only in a limited sense, as it is conventionally used to express that the condition must return true), since instances of primitive types are not objects, and hence, no dynamic proxies can be created for such values. The second condition expressed on the value of stock quotes in the ongoing example, though sound at compilation, could not be "registered" by a proxy, as the value of a quote is of primitive type. Even when attempting to create an instance of `Float` from the value returned by `getPrice()` and expressing the condition with the `compareTo()` method (Java's object counterpart to operators such as "<"), the interception chain is interrupted, since this method again returns a value of primitive type (float).

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

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

This sometimes comes as part of *operator overloading*, yet is not present in Java. The boxing and unboxing for primitive types included in Java at release 1.5 only covers automatic passing from primitive types to object types and vice versa, and does not transform operators to method invocations. In the case of `Obvents`, these shortcomings force programmers to think of possible comparisons, and to include these in the design of obvent types. In the case of the `StockQuote` class, could include a method `cheaperThan()` for expressing conditions on the price, as follows:

```
public class StockQuote... {
  private float price;
  ...
  public boolean cheaperThan(float than) {
    return price < than;
  }
}

Subscription<StockQuote> quoteSubscription = ...;
StockQuote formalQuote = quoteSubscription.constrain();
formalQuote.getCompany().equals("Telco");
formalQuote.cheaperThan(100.0);
quoteSubscription.activate();
```

We must, however, be aware that efficiency at the routing and filtering level might be reduced by doing so, as this method has to be executed, that is, corresponding obvents must be invoked and hence deserialized. This can become costly when performed several times in an overlay graph connecting publishers and subscribers. Java$_{PS}$ is in this sense clearly superior, since operators can be used *within* content filters.

6.3.2 *Beyond TPS—Supporting Uniform Remote and Local Programming.* As shown by TPS in Java, a complex type system potentially leads to many complications when deployed at a distributed scale. While primitive types might indeed be useful in certain strongly performance-sensitive applications, and the distinction between interfaces and classes definitely doesn't appear to be harmful in itself, these "irregularities" lead to different semantics and hence, require specific handling and implementations. They tend to represent special cases with respect to generics and reflection which are difficult to take into account, just like the possibility of directly manipulating fields (the main obstacle to the creation of dynamic proxies for classes in addition to interfaces). A *pure object-oriented type system* inherently enforcing type safety and also encapsulation at compilation, that is, avoiding direct field accesses (possibly by *unifying field accesses* and *method calls*, as in Eiffel [Meyer 1992b]), has the advantage of more easily supporting uniform interaction, even at a distributed scale.

If features such as primitive types are really required, effort should be invested in specific support for generics and reflection, possibly by fitting (e.g., by translation) these constructs into a more uniform underlying representation, and providing operator overloading. In Eugster [2006], for instance, we propose a first step in this direction through an extension to dynamic proxies in Java that supports both interfaces *and* classes. In order to ensure that proxy classes can be generated for classes by inheriting from these original classes and overriding their members, we introduce a *uniformly virtual object model* layer at which the runtime environment can override any member of any class.

The need for uniformity is particularly valid in the face of "modern" distributed programming abstractions, such as TPS (see also Section 7 for further cases), which tend to blend application logic with distribution-specific code. Since remote interaction introduces different semantics than local interaction, a lack of uniformity at a local scale already is likely to hamper safety when stepping to a distributed setting.

The next section extends the scope of this discussion.

7. DISCUSSION

In this section we discuss related distributed programming abstractions, and alternative programming language features for implementing distributed interaction through libraries [Eugster et al. 2004]. Finally, we compare the Java platform with Microsoft's .NET [Thai and Lam 2001] with respect to its support for the implementation of such abstractions.

7.1 Tuple Spaces

The tuple space abstraction was initially introduced in the Linda programming language [Gelernter 1985], in which these spaces served as the means of coordination between cooperating processes.

7.1.1 *Original Flavor.* Through a tuple space, processes can exchange arbitrary-length tuples of values. Inserting a tuple into the tuple space is done using the `out` primitive. Fetching a tuple from the tuple space is done using a blocking primitive, either `in` to subsequently remove the read tuple from the space or `read` to enable the same tuple to be read by several consumers (note that the tuple space abstraction has since been extended with further primitives, e.g., nonblocking primitives and callbacks on the consumer side).

Consider the following example expressed in Linda:

```
out ("StockQuote", "Telco", 100, 90.0);                    // 1
float f = 90.0;                                            // 2
in ("StockQuote", "Telco", 100, f);                        // 3
in ("StockQuote", "Telco", 100, var f);                    // 4
in ("StockQuote", "Telco", 100, g: float);                 // 5
```

In Line 1, a tuple consisting of four values is put into the tuple space. In Line 3, a tuple with four values is requested. Since the value of `f` is 90.0, the tuple from Line 1 matches the request, which means that this tuple may be extracted from the tuple space. The `var` keyword in Line 4 causes the `f` to be treated as a formal argument, that is, it can match any value. The tuple added in Line 1 may be extracted by Line 4, and the actual value of `f` would then be 90.0. In Line 5, `g` is declared as a variable and used as a formal argument like `f` in Line 4.

7.1.2 *Jada.* Implementing tuple spaces in Java poses similar problems to those described for TPS. As an example, Jada [Ciancarini and Rossi 1997] instruments Java with a library that supports tuple spaces. In Jada, class `Tuple` represents lists of Java `Objects`, and has constructors for up to ten values. Clients thus have to cast these objects explicitly upon reception, thereby reducing type safety. Formal arguments are represented by objects that stand for the desired type (instances of `java.lang.Class`). The aforementioned Linda example can be expressed in Jada as follows (Line 5 has no equivalent in Jada):

```
TupleSpace space = ...;
space.out(new Tuple(
  "StockQuote", "Telco", new Integer(100), new Float(90.0)));      // 1
float f = 90.0;                                                    // 2
Tuple tuple1 = space.in(new Tuple(
   "StockQuote", "Telco", new Integer(100), new Float(f)));        // 3
Tuple tuple2 = space.in(new Tuple(
  "StockQuote", "Telco", new Integer(100), Float.class));          // 4
f = ((Float)tuple2.getItem(4)).floatValue();
```

As illustrated by Jada, such a tuple space library becomes clumsy compared to the original support in the Linda language. A slight improvement can be

achieved with Java 1.5, which supports methods with a variable number of arguments. Using this feature, we can avoid declaring several constructors for different numbers of arguments, and simply define a single one:

```
public class Tuple {
  public Tuple(Object ... args) { ... args[i] ... }
}
```

Yet, this removes only a small portion of the problem, namely, on the implementation side. The elusive syntax remains for the instantiation of such tuples. In terms of type safety, no gain is observed.

7.1.3 *JavaSpaces.* More recent approaches to tuple space interaction in Java, such as JavaSpaces [Freeman et al. 1999], apply a different model, viewing tuples as single objects whose fields reflect tuple values. Custom events are defined by subtyping the basic `Entry` type, which again does not ensure type safety, since type-checks and type casts are necessary. Also, no encapsulation is provided by forcing fields to be declared `public`; expressing and performing any content-based filtering through these fields. More precisely, a given subscriber to 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 this subscriber is that `o` conforms to the dynamic 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.

With Javaspaces, stock quotes can be expressed as follows:

```
public class StockQuote extends Entry {
  public String company;
   public Integer amount;
   public Float price;
   public StockQuote(String company, Long amout, Float price) {
     this.company = company;
     this.amount = amount;
     this.price = price;
   }
}

JavaSpace space = ...;
space.write(new Stockquote(
   "Telco", new Integer(100), new Float(90.0)), ...);                    // 1
float f = 90.0;                                                          // 2
Entry entry1 = space.read(new StockQuote(
   "Telco", new Integer(100), new Float(f)), ...);                       // 3
Entry entry2 = space.read(new StockQuote(
   "Telco", new Integer(100), null), ...);                               // 4
f = ((StockQuote)entry2).price.floatValue();
```

7.1.4 *Satisfactory Library Implementation?.* The tuple space paradigm illustrates the difficulty of achieving the full features of a hardwired abstraction

with a library. Rather surprisingly, a separation of the programming language from the concurrency mechanism had been advocated by Carriero and Gelernter themselves earlier [Gelernter and Carriero 1992], while their Linda language is widely viewed as a monolithic solution to merging a coordination language with a programming language.

By type parameterizing the JavaSpace type, and changing the read() method so as to give access to a dynamic proxy, a similar "subscription" scheme to that used with TPS could be achieved. Through the official support for generics in Java 1.5, we could in fact expect a type parameterized version of the JavaSpace API to appear:

```
public interface JavaSpace<T extends Entry> {
  public T read(T t, Transaction txn, long timeout) throws ...
  public Lease write(T t, Transaction txn, long lease) throws ...
  ...
}
```

The template object-based matching would, however, lead to a potential mismatch between the type parameter and the type of template object with an extension such as that portrayed previously. Any aforementioned template object t is obviously an instance of a class type C, which is a subtype (distinct – see Section 3.3.1) of the type passed for T. In other words, though seemingly manifesting interest in all instances of the type passed for T, a subscriber will be actually only receiving instances of a more derived type given by the class C of the template object, but no instances of any class C' which subtypes T without being derived from C.

The distributed implementation of JavaSpaces could also strongly benefit from runtime information on actual type parameters (see Section 6.1) for setting up connections.

7.2 RMI

Despite the emergence of many contestants, the remote method invocation (RMI) paradigm (or remote procedure call, RPC) remains the most prominent abstraction for remote interaction in distributed object settings.

7.2.1 *Java RMI.* Java RMI relies on the inherent Java type system, yet further constrains the use of this type system in its own context: (1) Static types of remote references must be abstract types, that is, interfaces, and (2) any method in such an interface must imperatively declare RemoteException in its throws clause. Originally, Java RMI could be considered as a language add-on in the sense that a separate compiler (rmic) was used to generate type-specific proxies—the absence of corresponding proxies only being signalled at runtime. The original implementation of Java RMI could in this sense be viewed as an intermediate solution between a language-integrated RMI package and a standard library.

7.2.2 *Satisfactory Library Implementation?.* A form of RMI can be implemented in Java as a pure library (without specific compilation) with dynamic

proxies to defer the binding to a remote object to runtime. This proxy mechanism has, by all evidence, been devised with the requirements of Java RMI in mind, and in fact as of Java 1.5, is the preferred means of obtaining proxies for RMI, thus making the `rmic` precompiler obsolete. The absence of dynamic proxies for classes is, in the context of Java RMI, not a shortcoming, since by following the RMI specification, remotely invocable types are still to be declared as interfaces. However, an implementation of a *future*-like (*future type message passing* [Yonezawa et al. 1987]) asynchronous RMI for Java—where an invoking thread is not blocked, but immediately returned a handle for the result (with possible blocking when querying the handle prematurely, known as *wait-by-necessity* [Caromel 1993])—is not unproblematic. Indeed, to support *implicit* future invocations, where such handles are of the same type as the return values, types of return values of methods would have to be limited to abstract types in order to ensure that dynamic proxies could be returned as future objects. This would mean adding further constraints to the specification of Java RMI. Without these, a type parameterized interface `Future` defined in package `java.lang.reflect` must currently be used to convey return types of *explicit* future invocations, and building implicit futures on top of these requires complex static analysis [Pratikakis et al. 2004].

Providing both Java RMI and our TPS library side by side (including the possibility of passing around proxies as part of events) could lead to a powerful framework for programming distributed applications. By providing dynamic proxies for classes and possibly also primitive types, the TPS library could become more expressive, and implicit future invocations could be catered to, leading to much flexibility.

7.3 Join Patterns

More recently, Fournet and Gonthier proposed the *join pattern* abstraction in the context of their Join calculus [Fournet and Gonthier 1996] for concurrent and distributed computing. Join patterns were first put to work in the JoCaML language [Fournet et al. 1997] before more recently giving rise to an extension of C# [Hejlsberg and Wiltamuth 2001] called Polyphonic C# [Benton et al. 2004]. Our discussions here focus on the latter implementation of join patterns, as it is closer in syntax to Java.

7.3.1 *Chords.* For the declaration of join patterns (termed *chords* in Benton et al. [2004]), Polyphonic C# includes asynchronous methods. Such methods signal their absence of return values by the `async` tag in place of a return type, and return "immediately" by executing in a separate thread. Join patterns themselves each consist of a header and body. The header is composed of a set of body less method declarations (referred to as *pattern methods* in the following), separated by "&." Join pattern bodies are executed once *all* methods declared in the header have been called. Benton et al. [2004] describe the implementation of various predating abstractions for concurrent, but also distributed, programming, such as rendez-vous, active objects, and (asynchronous) request/reply, based on join patterns.

```
public class JoinPattern<T> {
  public class Pattern {
    public T registerMethod() throws AlreadyRegisteredException
      { /* return a dynamic proxy to register the sync patterns method */ }
    public T registerAsyncMethod()
      { /* return a dynamic proxy to register an async pattern method */ }
    public T registerBody() throws AlreadyRegisteredException
      { /* return a dynamic proxy to register the pattern body */ }
  }
  public T getInvocationProxy()
      { /* return a dynamic proxy to register an invocation */ }
  ....
}
```

Fig. 14. A library approach to join patterns in Java.

To better illustrate join patterns, consider the example of a buffer introduced in Benton et al. [2004], expressed in Java syntax:

```
public class Buffer {
  public String get() & async put(String s) {
    return s;
  }
}
```

Through the pattern expressed here, invocations to `get()` and `put()` are paired. If no execution of `put()` is pending, an invocation of `get()` is blocked until `put()` is invoked and the pattern body executes, returning the value of the string. The inverse does not lead to blocking `put()`, as this method is `async`hronous. Note, however, that this example, as pointed out in Benton et al. [2004], does not specify *which* invocations are paired: An invocation of `get()` following two unmatched `put()`s may return either value.

7.3.2 *Translating to Java.* Implementing the essential features of join patterns is possible in Java with generics and reflection features such as required by TPS. A corresponding abstraction is sketched by Figure 14 (details omitted for presentation).

Class `JoinPattern` defines an inner class `Pattern` for expressing single patterns. The latter class provides three methods enabling the specification of: (1) a synchronous pattern method (if any), (2) asynchronous pattern methods, and (3) a method representing the body of the corresponding pattern, respectively. For type safety, these are all expressed by invoking dynamic proxies returned by these methods. This requires that classes expressing join patterns define methods for all pattern methods used in pattern headers. Since the original join patterns use these pattern methods for synchronization, the bodies of these methods in our library implementation indicate their invocations to the corresponding pattern by simply forwarding them to a proxy, created through the `getInvocationProxy()` method. Concretely, the aforementioned `Buffer` defined would be translated as sketched in Figure 15.

7.3.3 *Satisfactory Library Implementation?.* Clearly, the `Buffer` class outlined in Figure 15 is far more complex than the variant based on specific

```
public class Buffer {
  static JoinPattern<Buffer> joins = new JoinPattern<Buffer>();
  static {
    /* describe the first join pattern */
    JoinPattern<Buffer>.Pattern pattern = joins.new Pattern();
    pattern.registerMethod().put(null);
    pattern.registerAsyncMethod().get();
    pattern.registerBody().putGetBody(null);
  }
  /* create a proxy to forward the invocations to the pattern */
  private Buffer proxy = join.getInvocationProxy();
  /* pattern methods for pattern header */
  public String get() { return proxy.get(); }
  public void put(String s) { proxy.put(s); }
  /* method body for the pattern */
  public String putGetBody(String s) { return s; }
}
```

Fig. 15. Buffer implemented with join patterns in Java.

constructs, yet achieves the same functionalities. More important are the shortcomings in terms of safety and performance. Indeed, only at runtime are errors detected, such as omitted invocation forwards in pattern methods, or attempts made to register more than one synchronous pattern method for the same pattern. Furthermore, just like in the case of the Adapter class in our library implementation of TPS, the JoinPattern class here would require runtime information regarding the type parameter. As regards performance, we would have to investigate whether the heavy use of reflection in the Pattern class would justify language primitives.

7.4 Alternative Support for Distributed Programming Abstractions

The prominent mechanisms discussed so far have all been investigated in the context of TPS in variants of Java. The intention is by no means to claim that the outlined mechanisms cover all possibilities, since there are many other languages and features to think about. As illustration, we discuss a small set of alternative language features.

7.4.1 *Closures.* In our library implementation of TPS, a subscriber implements a callback object that is passed to the adapter upon subscription. The code for such an obvent handler, that is, a class that implements Subscriber, is defined in a specific class. This is likely to lead to a scattering of the code related to single subscriptions unless making use of inner classes, as in our example in Section 5.3.

In our Java$_{PS}$ implementation, the preceding obvent handler is viewed as a closure whose signature is implicitly given as part of the syntax of the subscription expression, and all code related to a subscription is colocated, making it easy to understand what the subscription does. Given that the content filter and event handler are two sides of the same story, it seems more apropriate to concentrate these at the same place.

The closures used for expressing content filters have very specific semantics. As already mentioned in Section 4.2.3, these closures are characterized by deferred evaluation. To be fully accurate, these closures actually undergo

a deferred *compilation*, as they are transformed at compilation into instantiations of first class parse trees. This makes these closures "migratable" and transformable, for example, partially evaluatable.

Such a mechanism could in a more general sense be interesting for distributed programming. It could help implement pre- and postconditions (see Meyer [1992a]) efficiently in a distributed heterogenous RMI environment (e.g., by evaluating parts of preconditions on the client side before attempting any remote interaction), other forms of distributed filtering of objects passed by value between remote components, such as required by TPS previously, or even queries on object databases. Clearly, the expression of content filters reflects the need for future abstractions to seemlessly integrate with application code, and in TPS represents the most tedious part, with only few languages providing nearly adequate mechanisms.

7.4.2 *myType and Mixins.* The myType type qualifier was introduced by Bruce et al. [1995] in PolyTOIL, and inherited by Loom [Bruce et al. 1997]. In any given method body, this qualifier refers to the dynamic type of the considered object the type of this. In the words of the authors, "*myType is anchored to the type of the object in which it appears*." This paradigm enables an inherently clean implementation of binary methods. In the context of TPS, this could be used in combination with behavioral reflection and simple unbounded parametric polymorphism (also part of PolyTOIL) to ensure type safe *direct* subscriptions to application-defined event classes (e.g., without first-class adapter abstraction), which subtype a specific root obvent type, Obvent. Following the Java syntax, we could imagine having something like the following:

```
public class Obvent implements Serializable {
  public final void publish()
    throws CannotPublishException {...}
  public final Subscription<myType>
    subscribe(Subscriber<myType> s) {
    ...
  }
}

public class StockEventImpl extends Obvent ... {...}
```

Subscribing to an application-defined obvent class, such as the previous StockQuote class, can be done simply by first creating an instance of this class, and then invoking the subscribe() method (assuming the Subscriber class implementation follows that from Section 5.3):

```
Subscription<StockQuote> quoteSubscription =
new StockQuote().subscribe(new Subscriber<StockQuote>() {...});
StockQuote quote = quoteSubscription.constrain();
quotes.getCompany().equals("Telco");
...
```

The preceding subscription scheme could fulfill all our requirements in a language which, unlike Java, does not provide any purely abstract types (since these are not supported by the aforementioned design). Furthermore, if the myType type qualifier is available in class (static) methods, we could omit creating an instance of an event class just for subscribing to that type.

Similarly to myType, the concept of *mixin* [Bracha and Cook 1990] could enable merging the abstraction for subscribing with these very event types; here, by "adding" methods that express subscriptions and unsubscriptions to application-defined obvent types, instead of inheriting them from a root obvent type.

7.5 .NET

The experiences presented thus far have been mostly 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. Although, just like TPS, RPC relies on the invocation semantics and type systems 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 cause the invocation of coarse-grained remote, bound objects, but rather relies on the *transfer* of fine-grained, unbound objects, which might require transfering the code of such transferred objects, and possibly those of other objects. Interoperability in the case of TPS hence requires further assumptions, such as the implementation of obvent types in all involved languages (see OMG [2001a]), or a common intermediate programming language (e.g., byte code) with dynamic class loading and linking (see Section 6.2).

The claimed advantage of Microsoft's .NET platform over the Java suite is precisely the focus on programming language-independence. Though several programming languages can be compiled for deployment on Java virtual machines currently, .NET offers more inherent support for programming language heterogenity, for example, through reflection features or other core class libraries. Based on this fact, we have implemented our TPS library also for .NET [Baehni et al. 2003]. Definitely inspired by the historically preceding Java technology, .NET lends itself well to a side-by-side comparison with Java. It is hence interesting to investigate whether (besides in terms of interoperability), .NET is better qualified for distribution than Java, particularly in supporting demanding abstractions such as TPS.

From a general point of view, .NET indeed provides many mechanisms for distribution, with slightly more variants than Java. Merely for serializing objects, .NET provides three mechanisms (binary, SOAP, custom). Moreover, .NET *remoting*, the .NET counterpart to Java RMI, provides many means of configuring remote invocations. With respect to the previous shortcomings of Java, .NET can be summarized as follows:

—*Generics.* Generics were investigated in the context of .NET shortly after corresponding efforts for Java [Kennedy and Syme 2001]. The outcome was, however, only integrated into the common language runtime at release 2.0.

—*Reflection* .NET provides roughly the same introspection features as Java, and moreover provides dynamic proxies, albeit similarly restricted to interface types. However, .NET provides support for generating field access methods automatically.
—*Types.* Remotely invocable objects in .NET can also be of class types, and direct field accesses are automatically transformed to access method invocations. Since its very first release, .NET offers support for boxing and unboxing (here .NET seems to have preceded Java).

Hence, it appears that .NET has, from the type system point of view, introduced certain interesting support mechanisms. This is probably a consequence of the fact that it aims at providing language interoperability, and therefore has to be able to deal with various type systems. When considering reflection and generics from our TPS-biased perspective, .NET nonetheless still plays in the same league as Java.

8. RELATED WORK

This section presents related efforts, focusing on systems for event-based programming on one hand, and on programming languages for distributed programming on the other. In the former case, we emphasize the differences to our design principles for TPS outlined in Section 3.1, and in the latter case, we discuss the (differences to) the TPS model/abstraction.

8.1 Event Systems

Many sytems and models for distributed event-based programming have been described in the literature with some form of typed events. We outline some of the most prominent approaches.

8.1.1 *COM+.* Microsoft's COM+ [Oberg 2000] promotes a model that is based on the types of subscribers, rather than on the types of events: Similar to RPC, objects can provide specific interfaces defining the methods through which they can 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 values.

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

8.1.2 *CORBA.* Several event-based systems have been designed and specified for the common object request broker architecture (CORBA) [OMG 2002].

With the CORBA event service [OMG 2001b], 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 nonhierarchical subject names.

A form of typed interaction is provided, similar to the model in COM+, enabling the use of the types of not only consumers, but also producers (the CORBA event service supports pull- and push-style interaction) as the main subscription criterion. According to the type of interaction, methods only have input parameters or return values so as 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. Content-based subscriptions are, however, not supported.

Shortly after the first commercial implementations of the CORBA event service became available, several deficiencies (e.g., missing support for QoS and real-time requirements, difficulties with the aforementioned typed events) became apparent, leading to extended event service implementations. One of the most significant was that used in the TAO real-time ORB [Harrison et al. 1997]. It addresses mainly real-time issues, but also enforces subscriptions based on the identity of the publisher and/or event types. In the latter case, the "type" is an integer value explicitly assigned to every event by storing it in a dedicated property. This goes against **TS**, while the content-based subscription lacks support for **EP**.

Based on the observed deficiencies of the CORBA event service, the OMG issued a request for proposal of an augmented specification, the CORBA notification service [OMG 2000]. A *notification channel* is an event channel with additional functionalities, including notions of priority and reliability. A new form of semityped event, called *structured events*, was introduced. This represents general-purpose event types which manifest fields like event type and event name, and are roughly composed of an event header and event body. Both parts each consist of a fixed part and 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 that contradict both **AE** and **EP**. Standard properties include a notion of event type which is 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, thus contradicting **TS**.

8.1.3 *Java.* Just as for CORBA, several publish/subscribe schemes have been proposed for Java.

Sun's answer to the CORBA event and notification service specifications is the Java message service (JMS) [Happner et al. 2002]. 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, these consist simply of a type name. Content filters are expressed as *message selectors*, which are

essentially strings based on an SQL-like grammar. Hence, just like the CORBA notification service, JMS hampers **TS**, **EP**, and **AE**.

When consumers register callback objects with a JavaSpace, we end up with a publish/subscribe communication scheme in which JavaSpace plays the role of the event channel aimed at multicasting event notifications to a set of subscriber objects. As mentioned in Section 7.1.3, 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 this 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. JavaSpaces hence do not support **TS** and **EP**. As outlined in Section 7.1.3, these shortcomings could be avoided by making use of a predicate expression scheme, as in our TPS library implementation, as well as generics. By doing so, we could also relax the inherent limitation of a template-based matching scheme such as that promoted by JavaSpaces, which is the fact that only strict equality matching can be expressed by subscribers.

8.1.4 *CEA.* The Cambridge event architecture (CEA) [Bacon et al. 2000] 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, again these are based on viewing the events as sets of fields, hence contradicting **EP**.

8.2 Programming Languages

Many programming languages have been described with inherent support for distribution (and concurrency). We overview a select set of such languages, focusing on their abstraction(s) for distribution and the differences to TPS (or Java$_{PS}$), as well as on the use of reflection and generics for implementing these abstractions.

8.2.1 *Remote Objects.* Originally introduced for procedural programming models (e.g., *Sun RPC* [Srinivasan 1995], *DCE RPC* [Rosenberry et al. 1993]), remote invocations have been rapidly applied to object-oriented languages, promoting some form of remotely accessible entities, such as *guardians* in Argus [Liskov 1988] (follow-up of CLU [Liskov 1993]), or *network objects* in Modula-3 [Cardelli et al. 1989] and Obliq [Cardelli 1995] (and of course Java, as presented already in Section 7).

In Obliq, every object is potentially a network object. Objects which are declared `serialized` are objects on which no more than a single method can execute at a time. One of the distinguishing features of `protected` objects is that they can shield some of their methods from the "outside world" (e.g., physically remote hosts). The benefits of distinguishing between remote and local (views of) objects, and thereby introducing a form of granularity (large, location-bound objects versus smaller ones; see Section 3.2.1) have been retained by most subsequent languages—many through specific syntax like Obliq. In Eiffel, `separate`

objects are serialized in the sense of Obliq, and furthermore (potentially) under the control of a different thread of execution than the denoting objects [Meyer 2002]. The invocation of such an object then inherently leads to synchronization.

Both Obliq and Eiffel provide a notion of *conditional* execution of methods. In Obliq, *guards* are specific constructs added to this end, while the *contracts* inherent to Eiffel (more precisely, *preconditions*, in this case) are interpreted as wait conditions in certain cases, thereby conveying similar semantics as guards.

As discussed in Section 7, proxies such as those present in Java provide an ideal mechanism for remote invocations. Similarly, they can be used for expressing guards, just as they have been used for predicates in our library implementation of TPS. An inherent support for "distributed predicates," such as those found in Java$_{PS}$, could be generalized for expressing various forms of wait conditions (e.g., guards, preconditions).

8.2.2 *Asynchronous Objects.* Many concurrent programming languages build on the *actor* model [Agha 1985] in which actors (autonomous computational entities) communicate by asynchronous message passing. The behavior of such an actor consists essentially in handling one incoming message after another by reacting with some local computation, and possibly sending out further messages in conclusion. The semantics of the actor model has been thoroughly studied, and the model continues to be applied in the functional languages for which it was originally devised. An example of this is the Erlang programming language, which provides explicit asynchronous message sends/receives and further includes wait conditions [Armstrong et al. 1996].

In object-oriented languages, these asynchronous messages often appear as asynchronous *one-way* invocations, that is, asynchronous calls to methods without return values, which can thus be viewed as *unicast* events. Two early examples are Actalk [Briot 1989] and Act++ [Kafura et al. 1993].

Oz is a multiparadigm language which includes many features and paradigms for concurrent and distributed programming. Oz promotes the actor model, providing the concept of *mailboxes* for asynchronous message passing, and inherently supports wait conditions. Mobility is endorsed by the possibility of migrating computation [Havelka et al. 2004], and distribution is supported by leveraging on semantics of remote invocations [Grolaux et al. 2004].

Salsa is another more recent example of an actor-based language. Salsa builds on Java and proposes asynchronous (remote) message sends, introduced by "<-", side-by-side with synchronous (local) method invocations. The "@" operator is used explicitly for forcing ordering on multiple message sends that appear subsequently in the client code (e.g., leading to a FIFO ordering with respect to the client side).

Many object-oriented programming languages, even without being based on the actor model, have been augmented with a form of asynchronous invocation. An example is given by the aforementioned Eiffel model, which similarly includes asynchronous semantics for calls to methods without return values. For

method calls with return values, most languages proposing an idiom of asynchronous messaging are inspired by futures. As already outlined in Section 7.2, such asynchronous return values make a very strong case for dynamic proxies, just like any form of remote (asynchronous) invocation. In addition, proxies can also act as decorators, providing access to parameters for remote interaction (such as QoS) through an added interface.

8.2.3 *Mobile Objects.* Several languages aiming at mobility of objects (in the sense of mobile agents) have been founded on calculi. For instance, nomadic Pict [Unyapoth and Sewell 2001] builds on π calculus [Milner 1999] for concurrent computing, which similarly to the actor model, provides asynchronous message passing as the basic interaction paradigm. Nomadic Pict adds primitives for agent creation and migration, and location-independent communication of agents between sites. To achieve this location-independence, Nomadic Pict may use broadcast techniques, but these are not reflected at the language level.

Ambients [Cardelli and Gordon 1998] are another approach to mobile agents through an explicit notion of *location* and corresponding *boundaries*. Mobility is endorsed by giving ambients the ability to cross boundaries. Ambients also include a notion of security on the basis of these boundaries. Just like in Nomadic Pict, Ambients are presented without inherent support for broadcasting. A more recent proponent of similar guidelines underlying Ambients is AmbientTalk [Dedecker et al. 2006]. AmbientTalk provides both first-class future invocations and broadcast support. The integration of these features is facilitated by the absence of static typing; AmbientTalk is dynamically typed and object-based (prototype-based) rather than class-based, which pragmatically does away with issues of type agreement and conformance.

8.2.4 *Event Objects.* In contrast to the aforementioned languages, in which multicast interaction can be achieved by explicitly addressing multiple destinations, the ECO (events + constraints + objects) model [Haahr et al. 2000] is an approach to integrating event-based multicast interaction with a programming language akin to Java$_{PS}$. In ECO, which builds on C++, 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 constraint. *Notify constraints* are expressed based on the fields of events, and *preconstraints* as well as *postconstraints* use the state of the subscriber object. Methods cannot, however, be used to express constraints, as events do not include code. Hence, ECO does not provide **EP**. Various constraints found in the ECO model could be implemented with dynamic proxies (and generics) satisfying the requirements posed by TPS.

Prasad et al. (e.g., Ostrovsky et al. [2002]) have proposed several broadcast calculi. These come closer in spirit to Java$_{PS}$. Expressing Java$_{PS}$ in a broadcast calculus variant, bears the nontrivial problem of expressing content filters on broadcast channels. When mapping content-based channels (e.g., dissemination based on dynamic criteria) to named channels (e.g., static channels), several

assumptions are required to avoid explosion of complexity (see Opyrchal et al. [2000]).

9. CONCLUSIONS

We have presented type-based publish/subscribe (TPS), a high-level abstraction for anonymous, one-to-many interaction at large scale. Java$_{PS}$, our extension of the Java language, was motivated by the obvious lacks manifested by the Java language with respect to supporting the implementation of abstractions like TPS as libraries. To achieve some level of type safety with a library implementation, we started out by making use of a "future" version of Java incorporating generics. To enable a satisfactory expression of content filters, we explored the use of a recent reflection mechanism, resulting, however, in many restrictions.

In general, and in the face of today's heterogenity across platforms, we believe that programming languages should not be implemented with specific (support for) abstractions for distributed interaction as primitives. We rather believe that designers of future languages should foresee a more general support for distributed interaction abstractions. In particular, avoiding an integration avoids the intrinsically hard question of which abstractions should be supported.

Although TPS is surely not the last paradigm for distributed programming, the constraints imposed by TPS should be kept in mind when conceiving future support for distributed programming. As shown by the difficulty in expressing content filters, TPS, as a paradigm emphasizing scalability and performance, requires a strong interaction with the native programming language, and is hence a very demanding abstraction. Most abstractions established for distributed interaction, such as tuple spaces or RMI, probably require only a subset of the features mandated by TPS.

We argue that reflection and generics, as faces of *extensibility*, are key concepts for a general language support of distributed programming, and that a straightforward type system can support the implementation of such features. With inherent reflective capabilities and generics, we believe it is possible to implement a powerful TPS library, and, as pointed out in this article, also alternative abstractions for distributed interaction, such as tuple spaces, RMI, and even a more recent paradigm, join patterns.

Other authors have pointed out that extensibility of an object-oriented language requires generics (e.g., Steele [1999]) and reflection, and as a result, generics have been recently added to Java. In this article, we have identified through TPS a precise case for this argument in the area of distributed programming. We have illustrated how this case poses more stringent demands on programming languages than those previously expressed and partially addressed without distribution in mind. We have also argued, through Java, that current features in mainstream languages are still not sufficient for distributed programming.

We insist on the fact that in the face of modern abstractions for distributed interaction such as TPS, generics need to be provided in a form that includes runtime support for type parameters, and that reflection has to go beyond simple message reification (considered sufficient in the context of RMI, e.g., Aksit

et al. [1993]). We pointed out that from our perspective, the current support in Java for generics and reflection is clearly insufficient, and we illustrated how primitive types as well as direct field accesses contribute to these deficiencies. Last but not least, we have had a look at how these features map to Microsoft's .NET platform [Thai and Lam 2001]. Inspired by Java, .NET proposes closely related concepts of generics, as well as reflection, with nearly the same limitations. For instance, field accesses cannot be intercepted, which is, however, counterbalanced by the fact that types in .NET languages such as C# can declare properties, a form of fields with inherent support for getter/setter methods.

ACKNOWLEDGMENTS

We are very grateful to Agilent Laboratories, Lombard Odier Darier Hentsch and Co., the Swiss Group for Object-Oriented Programming, and the Swiss National Science Foundation for financially supporting efforts leading to the results presented in this article. Many thanks go also to the following individual contributors for fruitful discussions and invaluable feedback: Sébastien Baehni, Christian Damm, Rachid Guerraoui, Sidath Handurukande, Martin Odersky, Manuel Oriol, and Joe Sventek. Last but not least, we would like to thank the various reviewers which have invested time in commenting on this work.

REFERENCES

AGHA, G. 1985. Actors: A model of concurrent computation in distributed systems. Ph.D. thesis, University of Michigan, Computer and Communication Science.

AGUILERA, M. AND STROM, R. 2000. Efficient atomic broadcast using deterministic merge. In *Proceedings of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 209–218.

AGUILERA, M., STROM, R., STURMAN, D., ASTLEY, M., AND CHANDRA, T. 1999. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 53–62.

AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1993. Abstracting object interactions using composition filters. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*. 152–184.

ALTHERR, M., ERZBERGER, M., AND MAFFEIS, S. 1999. iBus—A software bus middleware for the Java platform. In *Proceedings of the International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS)*. 43–53.

ARAUJO, F. AND RODRIGUES, L. 2002. On QoS-Aware publish/subscribe. In *Proceedings of the International Workshop on Distributed Event-based Systems (DEBS)*.

ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. 1996. *Concurrent Programming in Erlang*, 2nd ed. Prentice-Hall, Upper Saddle River, NJ.

BACON, J., MOODY, K., BATES, J., HAYTON, R., MA, C., MCNEIL, A., SEIDEL, O., AND SPITERI, M. 2000. Generic support for distributed applications. *IEEE Comput. 33*, 3 (Mar.), 68–76.

BAEHNI, S., EUGSTER, P., AND GUERRAOUI, R. 2002. OS support for peer-to-peer programming: A case for TPS. In *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 355–362.

BAEHNI, S., EUGSTER, P., AND GUERRAOUI, R. 2004. Data-Aware multicast. In *Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks (DSN)*. 233–242.

BAEHNI, S., EUGSTER, P., GUERRAOUI, R., AND P.ALTHERR. 2003. Pragmatic type onteroperability. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*.

BENTON, N., CARDELLI, L., AND FOURNET, C. 2004. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst. 26*, 5 (Sept.), 769–804.

BIRMAN, K. 1993. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec.), 36–53.

BRACHA, G. 2004. Generics in the Java programming language. Tech. Rep., Sun Microsystems, Inc. July.

BRACHA, G. AND COOK, W. 1990. Mixin-Based inheritance. In *Proceedings of the 5th ACM Conference on Object-Oriented Programming Systems, Languages and Applications and 4th European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*. 303–311.

BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 1998. 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)*. 183–200.

BRIOT, J.-P. 1989. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP)*. 109–129.

BRUCE, K., PETERSEN, L., AND FIECH, A. 1997. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*. 104–127.

BRUCE, K., SCHUETT, A., AND VAN GENT, R. 1995. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*. 27–51.

CANNING, P., COOK, W., HILL, W., OLTHOFF, W., AND MITCHELL, J. 1989. F-Bounded polymorphism for object-oriented programming. In *Proceedings of the 4th ACM International Conference on Functional Programming and Computer Architecture (FPCA)*. 273–280.

CARDELLI, L. 1995. A language with distributed scope. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 286–297.

CARDELLI, L., DONAHUE, J., JORDAN, M., KALSOW, B., AND NELSON, G. 1989. The modula-3 type system. In *Conference Record of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 202–212.

CARDELLI, L. AND GORDON, A. 1998. Mobile ambients. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*. 140–155.

CAROMEL, D. 1993. Towards a method of object-oriented concurrent programming. *Commun. ACM 36*, 90–102.

CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. 2000. Achieving scalability and expressiveness in an Internet-Scale event notification service. In *Proceedings of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 219–227.

CHAMBERS, C. 1995. The cecil language specification and rationale: Version 2.0. Tech. Rep. UW-CS 93-03-05, Department of Computer Science and Engineering, University of Washington. Dec.

CIANCARINI, P. AND ROSSI, D. 1997. Jada—Coordination and communication for Java agents. In *Mobile Object Systems: Towards the Programmable Internet*. Lecture Notes in Computer Sceince, vol. 1222. Springer, 213–228.

DAMM, C., EUGSTER, P., AND GUERRAOUI, R. 2004. Linguistic support for distributed programming abstractions. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 244–251.

DEDECKER, J., CUTSEM, T. V., MOSTINCKX, S., D'HONDT, T., AND MEUTER, W. D. 2006. Ambient-Oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*. 230–254.

DELPORTE-GALLET, C., FAUCONNIER, H., GUERRAOUI, R., AND KOUZNETSOV, P. 2005. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.* 65, 492–505.

ELLIS, M. AND STROUSTRUP, B. 1992. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA.

ENGLER, D., WALLACH, D., AND KAASHOEK, M. 1996. Design and implementation of a modular, flexible, and fast system for dynamic protocol composition. Tech. Rep. TM-552, Massachusetts Institute of Technology, Laboratory for Computer Science. May.

EUGSTER, P. 2006. Uniform proxies for Java. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 139–152.

EUGSTER, P., DAMM, C., AND GUERRAOUI, R. 2004. Towards safe distributed application development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. 347–356.

EUGSTER, P., FELBER, P., GUERRAOUI, R., AND HANDURUKANDE, S. 2002. Event systems: How to have ones cake and eat it too. In *Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS)*. 625–630.

EUGSTER, P. AND GUERRAOUI, R. 2001. Content-Based publish/subscribe with structural reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS)*. 131–146.

EUGSTER, P. AND GUERRAOUI, R. 2002. Probabilistic multicast. In *Proceedings of the 3rd IEEE International Conference on Dependable Systems and Networks (DSN)*. 313–323.

EUGSTER, P., GUERRAOUI, R., AND DAMM, C. 2001. On objects and events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 131–146.

EUGSTER, P., GUERRAOUI, R., HANDURUKANDE, S., KERMARREC, A.-M., AND KOUZNETSOV, P. 2003. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst. 21*, 4 (Nov.), 341–374.

EUGSTER, P., GUERRAOUI, R., AND SVENTEK, J. 2000. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. 252–276.

FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr.), 217–246.

FOURNET, C. AND GONTHIER, C. 1996. The reflexive chemical abstract machine and the join calculus. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 372–385.

FOURNET, C., LANEVE, C., MARANGET, L., AND REMY, D. 1997. Implicit typing à la ML for the join-calculus. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR)*. 196–212.

FREEMAN, E., HUPFER, S., AND ARNOLD, K. 1999. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, Reading, MA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.), 80–112.

GELERNTER, D. AND CARRIERO, N. 1992. Coordination languages and their significance. *Commun. ACM 35*, 2 (Feb.), 97–107.

GOLDBERG, A. AND ROBSON, A. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification*, 2nd ed. Addison-Wesley, Reading, MA.

GREGONO, P. AND SAKKINEN, M. 2000. Copying and comparing: Problems and solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. 226–250.

GROLAUX, D., GLYNN, K., AND ROY, P. V. 2004. A fault tolerant abstraction for transparent distributed programming. In *the 2nd International Conference on Multiparadigm Programming in Mozart/Oz (MOZ)*. 149–160.

HAAHR, M., MEIER, R., NIXON, P., CAHILL, V., AND JUL, E. 2000. Filtering and scalability in the ECO distributed event model. In *Proceedings of the 5th IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE)*. 83–92.

HAPPNER, M., BURRIDGE, R., AND SHARMA, R. 2002. Java message service. Tech. Rep., Sun Microsystems, Inc. Mar.

HARRISON, T., LEVINE, D., AND SCHMIDT, D. 1997. 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)*. 184–200.

HAVELKA, D., SCHULTE, C., BRAND, P., AND HARIDI, S. 2004. Thread-Based mobility in oz. In *the 2nd International Conference on Multiparadigm Programming in Mozart/Oz (MOZ)*. 137–148.

HEJLSBERG, A. AND WILTAMUTH, S. 2001. *C# Language Specification*. Microsoft Press, Redmond, WA.

KAFURA, D., MUKHERJI, M., AND LAVENDER, G. 1993. ACT++: A class library for concurrent program. in C++ using actors. *J. Object Oriented Program. 6*, 6 (Oct.), 47–55.

KENNEDY, A. AND SYME, D. 2001. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

KOENIG, P. 1999. Messages vs objects for application integration. *Distrib. Comput. 2*, 3 (Apr.), 44–45.

KRISHNAMURTHY, B. AND ROSENBLUM, D. 1995. Yeast: A general purpose event–action system. *IEEE Trans. Softw. Eng. 21*, 10 (Oct.), 845–857.

LIEBERMANN, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 214–223.

LISKOV, B. 1988. Distributed programming in argus. *Commun. ACM 31*, 3 (Mar.), 300–312.

LISKOV, B. 1993. A history of CLU. *ACM SIGPLAN Not. 28*, 3 (Mar.), 133–147.

MANSOURI-SAMANI, M. AND SLOMAN, M. 1997. GEM: A generalized event monitoring language for distributed systems. *Distrib. Syst. Eng. 4*, 2 (June), 96–108.

MEYER, B. 1992a. Applying design by contract. *IEEE Comput. 25*, 10 (Oct.), 40–51.

MEYER, B. 1992b. *Eiffel: The Language*. Object-Oriented Series. Prentice-Hall, Upper Saddle River, NJ.

MEYER, B. 2002. Systematic concurrent object-oriented programming. *Commun. ACM 36*, 9, 56–80.

MILNER, R. 1977. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MILNER, R. 1999. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, New York.

NIELSON, F. AND NIELSON, H. 1988. Two-Level semantics and code generation. *Theor. Comput. Sci. 56*, 1 (Jan.), 59–133.

OBERG, R. 2000. *Understanding and Programming COM+*. Prentice Hall, Upper Saddle River, NJ.

OKI, B., PFLUEGL, M., SIEGEL, A., AND SKEEN, D. 1993. The information bus—An architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*. 58–68.

OMG. 2000. *Notification Service Standalone Document*. OMG.

OMG. 2001a. *The Common Object Request Broker: Architecture and Specification, Chapter Value Type Semantics*. OMG.

OMG. 2001b. Event service. In *CORBAservices: Common Object Services Specification*. OMG.

OMG. 2002. *The Common Object Request Broker: Architecture and Specification, Version 3.0*. OMG.

OMG. 2003. *Data Distribution Service for Real Time Systems Specification*. OMG.

OPYRCHAL, L., ASTLEY, M., AUERBACH, J., BANAVAR, G., STROM, R., AND STURMAN, D. 2000. Exploiting IP multicast in content-based publish-subscribe systems. In *Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*. 185–207.

OSTROVSKY, K., PRASAD, K., AND TAHA, W. 2002. Towards a primitive higher order calculus of broadcasting systems. In *Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. 2–13.

POWELL, D. 1996. Group communications. *Commun. ACM 39*, 4 (Apr.), 50–97.

PRATIKAKIS, P., SPACCO, J., AND HICKS, M. 2004. Transparent proxies for Java futures. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 206–223.

RIVARD, F. 1996. Smalltalk: A reflective language. In *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection)*. 21–38.

ROSENBERRY, W., KENNEY, D., AND FISHER, G. 1993. *OSF Distributed Computing Environment: Understanding DCE*. O'Reilly, Sebastopol, CA.

SOLORZANO, J. AND ALAGIC, S. 1998. Parametric polymorphism for Java: A reflective solution. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 216–225.

SRINIVASAN, R. 1995. RFC 1831: Remote procedure call protocol specification version 2. Tech. rep., Sun Microsystems, Inc. Aug.

STEELE, G. 1999. Growing a language. *Higher-Order Symb. Comput. 12*, 3 (Oct.), 221–236.

SUN. 2005. *Core Java J2SE 5.0*. Sun Microsystems, Inc. http://java.sun.com/j2se/1.5.0/.

TAHA, W. AND SHEARD, T. 1997. Multi-Stage programming. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. 321–321.

TALARIAN CORPORATION. 1999. Everything you need to know about Middleware: Mission-Critical interprocess communication (white paper). Talarian Corporation, http://www.talarian.com/.

THAI, T. AND LAM, H. 2001. *.NET Framework Essentials*. O'Reilly, Sebastopol, CA.

THOMAS, D. 2004. Message oriented programming: The case for first class messages. *J. Object Technol. 3*, 5 (May–June), 7–12.

TIBCO. 1999. TIB/Rendezvous white paper. TIBCO, Inc. http://www.rv.tibco.com/.

UNYAPOTH, A. AND SEWELL, P. 2001. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Conference Record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 116–127.

YONEZAWA, A., SHIBAYAMA, E., TAKADA, T., AND HONDA, Y. 1987. 4: Modeling and programming in an object-oriented concurrent language abcl/1. In *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, MA. 55–89.

ZENGER, M. AND ODERSKY, M. 2001. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*.

Received February 2005; revised November 2005; accepted July 2006