OS Support for P2P Programming: a Case for TPS

Sébastien Baehni[1], Patrick Th. Eugster, Rachid Guerraoui
Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne

Abstract

Just like Remote Procedure Call (RPC) turned out to be a very effective OS abstraction in building client-server applications over LANs, Type-based Publish-Subscribe (TPS) can be viewed as a high-level candidate OS abstraction for building Peer-to-Peer (P2P) applications over WANs.

This paper relates our preliminary, though positive, experience of implementing and using TPS over JXTA: an analogous to the sockets for P2P infrastructures. We show that, at least for P2P applications with the Java type model, TPS provides a high-level programming support that ensures type safety and encapsulation, without hampering the decoupled nature of these applications. Furthermore, the loss of flexibility (inherent to the use of any high level abstraction) and the performance overhead, are negligible with respect to the simplicity gained by using TPS.

Keywords: Peer-to-Peer, Publish/Subscribe, Distributed Operating Systems, Object Oriented Programming, Events.

Technical Areas: Operating Systems, Web Computing, Middleware, E-Commerce.

1. Introduction

Remote Procedure Call (RPC) was first proposed by Birrel and Nelson [BN83] as a simple abstraction that conceals interactions between remote components beneath traditional procedural interfaces. Partly because of its simplicity and the very facts that it preserves object encapsulation and ensures type safety, and partly because RPC's overhead was very acceptable over sockets, RPC became a dominant paradigm for programming distributed applications over client/server architectures. In these architectures, clients typically communicate with one or several servers following a strongly-coupled request/reply scheme.

With the emergence of Peer-to-Peer (P2P) infrastructures, new forms of decoupled (i.e., anonymous and asynchronous) interactions are needed. One can indeed extend RPC with decoupling flavours. Nevertheless, regardless of the fact that adding layers over RPC would certainly hamper performance, it is

1. Contact e-mail: Sebastien.Baehni@epfl.ch

challenging to devise a simple abstraction that could directly fit these architectures and potentially be supported by future Internet-wide operating systems.

So far, P2P developers have generally gravitated towards a few application types: instant messaging (ICQ, AOL's Instant Messenger); collaboration (Aimster, Groove Networks); searching and file sharing (Morpheus, AudioGalaxy); distributed computation (Seti@Home, Parabon). Going beyond these simple applications, and developing more advanced ones, goes through developing basic abstractions for P2P programming.

Some initiatives were recently made towards building libraries or frameworks for deploying P2P applications. A seminal example is the JXTA [SUN01] specification whose implementations provide, for example, protocols for service discovery and many-to-many communication. This specification is rather low level and its protocols can be viewed as the analogous of the basic TCP or UDP protocols [Tan96] for client/server programming over sockets: one needs to explicitly cast types and control encapsulation.

Just like RPC typically hides the underlying mechanisms of sockets and preserves type safety and encapsulation, Type-Based Publish/Subscribe (TPS) [EGD01], a variant of Publish/Subscribe [OPS+93], can be viewed as a reasonable candidate abstraction to hide the mechanisms of a low-level P2P library, like JXTA. The distributed event-based interaction scheme promoted by TPS enables the preservation of the decoupled flavor of P2P applications.

This paper presents an implementation of TPS over JXTA and compares the programming and the performance of TPS in writing a typical P2P application with the programming and the performance of using directly JXTA in writing the very same P2P application. Our implementation of TPS over JXTA together with our performance comparisons provide a preliminary, yet interesting, experience towards evaluating the feasibility of equipping future Internet-wide operating systems with abstractions like TPS.

Like any high level abstraction, TPS does not apply to all kinds of applications and is obviously less flexible than a lower level library like JXTA. In particular, JXTA simply assumes a common XML knowledge among peers, denoting a very high interoperability, whereas the current implementation of TPS restricts to applications that share the common Java type model. We show that for these applications, the inherent benefits of the use of our TPS library, namely type-safety, encapsulation of application defined event types and code reusability, can be provided without hampering the decoupled nature of P2P computing.

This paper is organized as follows. Section 2 is a brief tutorial on JXTA. Section 3 describes a TPS API and an implementation of this API over JXTA. Section 4 compares the programming of an application using TPS and directly using JXTA. Section 5 compares the performance of these two implementations. Section 6 summarizes and concludes our experience.

For presentation simplicity and space limitations, we only give excerpts of the interfaces and classes of our implementations. The complete code of our TPS implementation and our testbed applications (both using TPS and directly using JXTA) are available at: http://lpdwww.epfl.ch [LPD01].

2. Background: JXTA

We recall here the basics of JXTA, on top of which we built our TPS abstraction layer. JXTA is a library specification for P2P computing, defining three layers: a core layer, a service layer and an application layer. The application layer wraps all the applications that are developed by JXTA programmers. The service layer is made up of services simplifying the development of the programmer. Various services are currently being implemented by the JXTA community; the best known are the monitoring service, the cms (content management system) service and the wire service (responsible for providing many-to-many communication). The core JXTA layer consists of several protocols ensuring basic communication between the peers, message routing or peer group creation.

2.1 The concepts

The JXTA protocols rely on six concepts: *ID*, *Peer*, *Pipe*, *PeerGroup*, *Advertisement* and *Message*. An ID identifies any JXTA resource, which can be a peer, a pipe, a peergroup or a codat (code and data). The peer concept points out all networked devices using JXTA. Any device with an electronic pulse is a JXTA peer (refrigerator, PDA, computer, ...). There are different kinds of peers: “normal” ones and ones that have additional functionalities. Rendez-vous (rdv) are specific peers that keep track of information about peers that are connected. Rendez-vous allow to make the bridge between two different subnetworks. They are mainly used to dispatch information and discovery queries between peers. The second kind of special peers are routers. These are used to route the information from one peer to another if they cannot communicate directly. Peers may have multiple network interfaces.[1]

In order for the peers to communicate, they need a mechanism that does not depend on their network. This mechanism is the pipe. A pipe is a virtual communication channel used to send messages. The basic pipes are asynchronous and uni-directionnal but some other variants are available (e.g., the very new bi-directional pipes or the many-to-many pipes (called wire)). Pipes are not bound to any physical address (like IP ones). Hence if a peer changes its address, it can continue to use the same pipe for sending or receiving messages. PeerGroups are collections of peers. A peer may join multiple peergroups to share different resources and services. There is no hierarchy inside the groups. A peergroup creates a scoped and monitored environment.

When a new resource (peer, pipe, peergroup, service) is available, a new advertisement is published in order for the other peers to know this ressource. An advertisement is a XML message that provides information about the resource. A typical peer advertisement would give information about the network interfaces it provides, about which groups it belongs to, about its name and ID. Each advertisement encompasses an age to distinguish stale advertisements from new ones.

1. For example, a peer may be able to send messages using TCP, IP-Multicast, HTTP, BlueTooth, BEEP, etc.... It allows different kinds of peers to communicate with each other using routers or rdv if they cannot talk directly to each other (i.e., because they do not have the corresponding network interfaces).

2.2 The protocols

Implementing the JXTA specification consists in implementing the following protocols: Peer Discovery Protocol (PDP, Figure 1), Peer Resolver Protocol (PRP, Figure 2), Peer Information Protocol (PIP, Figure 3), Peer Membership Protocol (PMP, Figure 4), Pipe Binding Protocol (PBP, Figure 5) and Endpoint Routing Protocol (ERP, Figure 6).

The PDP allows different peers to find each other. In fact, this protocol allows to find any kind of published advertisements. Without this protocol, a peer remains alone unless it knows in advance the peers it wants to connect to. This protocol uses the rdv/router peers to improve its performance and the PRP to achieve different discoveries.

The PRP is a protocol just above the transport layer. This protocol dispatches each JXTA message to the right services. The more handlers are registered with PRP, the more peers a given peer is potentially able to communicate with.

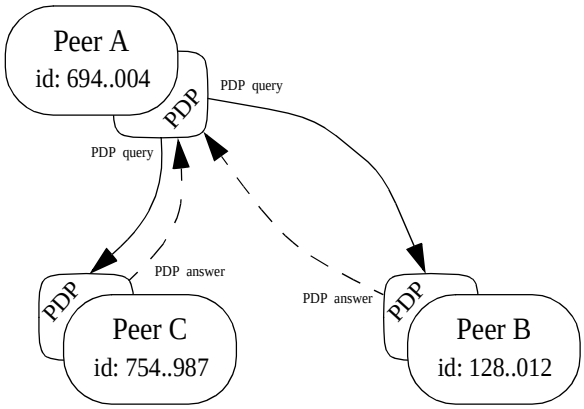

Fig. 1: Basic view of the PDP

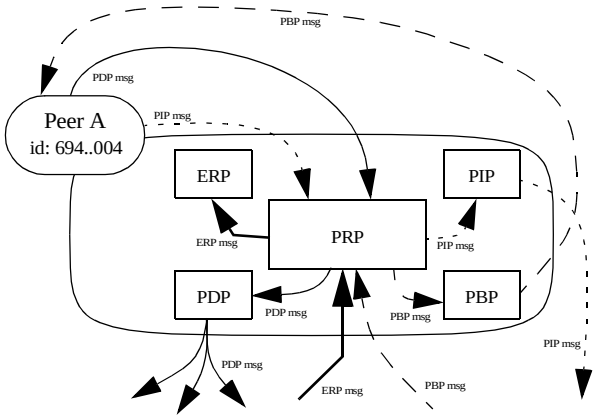

Fig. 2: Basic view of the PRP

The PIP is used to know the status of a peer. This protocol is responsible for finding and dispatching information about a peer, like the time the peer was up, the different incoming and outgoing channels, the traffic on them, and the different target and source IDs.

The PMP is used to obtain information about group membership requirements (credentials, password requirements, ...). Once a peer has those requirements, it can apply for membership as well as it can leave and join the group. This protocol is also used to update and cancel the membership, or create a secure environment using different credential authentification protocols.

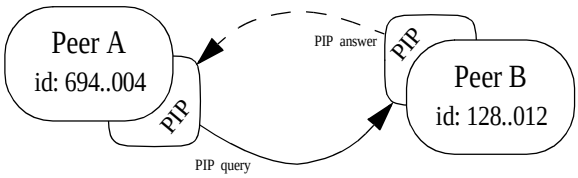

Fig. 3: Basic view of the PIP

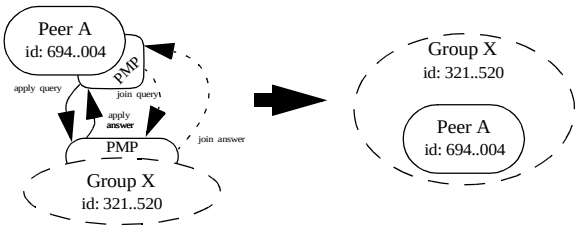

Fig. 4: Basic view of the PMP

The PBP is responsible for keeping the different peers of a pipe bound together. Even if the peers are moving in the network (i.e., if their IP addresses do not remain the same), they can continue to use the same pipes to send/receive messages.[1]

The ERP is used to route the different messages between the different peers. This allows different peers to exchange messages even when they do not know how to connect to each other (because of a firewall for example).

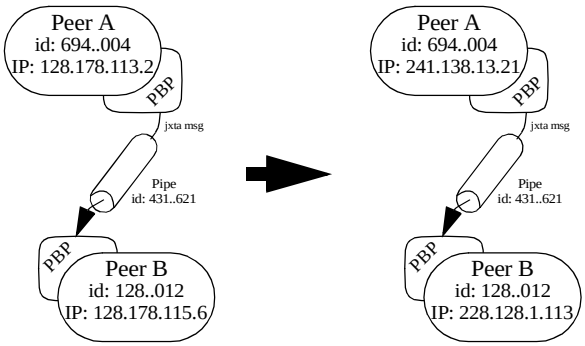

Fig. 5: Basic view of the PBP

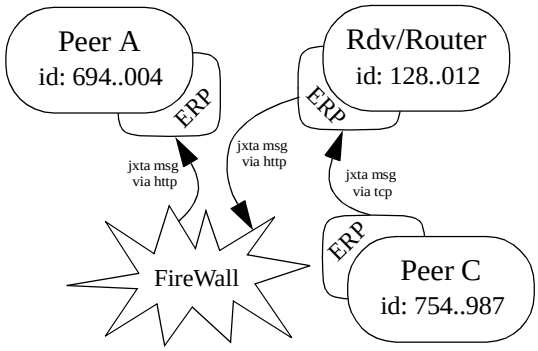

Fig. 6: Basic view of the ERP

3. TPS over JXTA

This section overviews the design and implementation of our TPS abstraction over JXTA.

3.1 TPS: Overview

The publish/subscribe paradigm is a communication pattern that provides *time*, *space* and *flow* decoupling among communicating entities. More precisely, the publishers and the subscribers (a) do not need to be up at the same time (time decoupling), (b) do not need to know each other (space decoupling) and (c) the sending/receiving of messages does not block the participants (flow decoupling). This paradigm perfectly suits decoupled networks and serverless architectures. In the original pattern (e.g., [TIB99]), publishers publish information on a subject and subscribers subscribe to subjects . These different subjects are often arranged in hierarchies (specified by a URL-like notation). More advanced communication schemes can be obtained through content-based subscribing, where subscribers express interests in events with particular native properties (e.g., [Gryp01]). In our Type-based Publish/Subscribe (TPS) scheme (see Figure 7), the subject is the event object type and the content is the state of instances of that type. Moreover, TPS ensures type safety and preserves event encapsulation with application-defined event types: the subscriber knows in advance the type of events it receives (type-safety) and subscriptions operations of the type can be used for content-based filtering (encapsulation). So one can easily implement content-based publish/subscribe (hence subject-based) using TPS.

1. To achieve that, the protocol uses the IDs of the peers. In fact, instead of counting upon a fixed IP address, the protocol relies on a fixed Universal Unique IDentifier (UUID) for each peer. So if the peer crashes and comes up again, it can find the peer it was communicating with and can continue to do so even if the IP address is not the same.

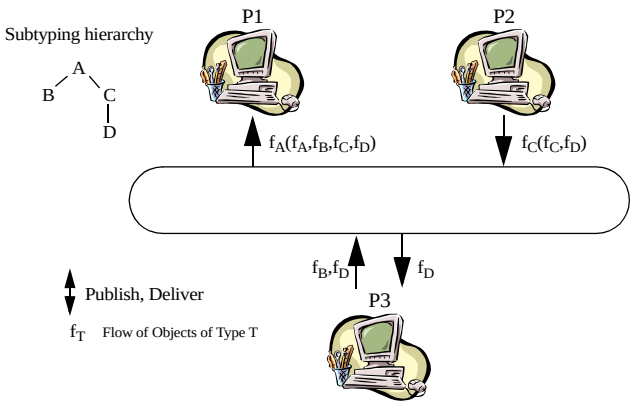

Fig. 7: Type-Based Publish/Subscribe (TPS)

3.2 Generic Java

Our TPS implementation we relate in this paper is based on *genericity*. Using TPS for a specific type T can be viewed as using instances of generic classes with a type parameter instantiated with T. Such generic classes are supported by several languages like C++ (`template`) and Ada (`generic`), while Java supports generics by the idiom of replacing variable types by the top of the type hierarchy. For such languages lacking generic types and methods, adequate extensions have been widely studied. In the case of Java, several solutions have been proposed like Generic Java (GJ) [BOSW98] which we have used for our implementation.

Our implementation uses the 1.3 version of the 14th Java Specification Request (JSR), based on GJ, which is expected to be included in the 1.5 version of Java.[1]

3.3 The TPS API

The different methods a programmer can use to express a TPS interaction are regrouped within our `TPSInterface`. The corresponding source code is given in Figure 8.

We briefly describe below each of these methods:[2]

`(1)`: This method is used to publish an instance of a type (`Type`) which can be any application-defined type. This instance is sent as an event to the subscribers.

`(2)`: This method (as the next one, i.e., `(3)`) is used to subscribe to the events of a specific `Type`. Two parameters must be provided: (a) a call-back object which is used to handle received events and (b) a handler for the exceptions that may be raised while handling the received events.

`(3)`: This alternative subscription method is used to register several call-back objects to handle the events in different ways. It is very useful, for instance, if we want to display the complete description of the events in a console and have a sketch of them in a GUI at the same time (for example, see Figure 12 and Figure 13).

1. The 1.3 version of the 14th JSR is a fully Java compatible compiler and enables the use of the original Java Virtual Machine (JVM).
2. Methods 1,2,3 and 4 could throw a publish/subscribe exception (*PSException*). We do not discuss these exceptions here, see http://lpdwww.epfl.ch for details [LPD01].

(4): This method is used to unsubscribe a specified call-back object and its associated exception handler. By doing so, only the specified call-back object is removed.

(5): This method is used to remove all the call-back objects registered so far. After this call, no event is received anymore.

(6, 7): The last two methods are used to obtain the entire set of events received or sent so far.

The other type of the API that the programmer needs to handle is TPSEngine. This class gives a reference to the TPSInterface. Here is the sketched source code of the TPSEngine class:

```
public class TPSEngine<Type> {
   public TPSEngine() {...}
   public TPSInterface newInterface(String name, Criteria c, Type t, String[] arg) {...}
}
```

The programmer uses this class in the initialization phase to get the TPSInterface.

```
public interface TPSInterface<Type> {
    public void publish(Type type) throws PSException;                        // (1)
    public void subscribe(TPSCallBackInterface<Type> tpsCBI,
                        TPSExceptionHandler<Type> tpsExH) throws PSException;  // (2)
    public void subscribe(TPSCallBackInterface<Type>[] tpsCBI,
                        TPSExceptionHandler<Type>[] tpsExH) throws PSException;// (3)
    public void unsubscribe(TPSCallBackInterface<Type> tpsCBI,
                          TPSExceptionHandler<Type> tpsExH) throws PSException;// (4)
    public void unsubscribe();                                                // (5)
    public Vector objectsReceived();                                          // (6)
    public Vector objectsSent();                                              // (7)
}
```

Fig. 8: The TPSInterface

3.4 Architecture

The TPS layer fits between the application layer and the JXTA layer. In our architecture (Figure 9, Figure 10 and Figure 11), one type is represented by one advertisement. When a subscriber subscribes to a type, it must specify an object implementing the TPSCallBackInterface for that type to handle the events and an TPSExceptionHandler (see Section 4.3.3 for an implementation example) responsible for handling the exceptions that may occur while dispatching the events. Our TPS layer is made up of four building blocks (see Figure 10):

- TPSEngine: This block is the core of our service. It collects and dispatches the subscriptions and publications.
- Advertisements (Advs): This block is responsible for creating a new advertisement for the type we are interested in as well as for finding and collecting the multiple advertisements that are in relation with our type.
- Interface Repository (IR): This block stores all the call-back interfaces and exception handlers. It also starts and stops the subscriptions
- Connections: This block creates readers, input pipes and output pipes from an advertisement. It sends and receives new messages with the underlying JXTA-WIRE service.

Our `JxtaTPSEngine` class implements the `TPSInterface` (see Section 3.3). It collects the publications and subscriptions and dispatches them to the `TPSAdvertisementsManager`. The latter instanciates all the other classes and dispatches the publications and subscriptions to the corresponding ones. The `AdvertisementsCreator` is responsible for creating an advertisement (see Section 4.4.1) and the `TPSAdvertisementsFinder` (see Section 4.4.2 for a non-generic version of it) searches new advertisements and dispatches them to the registered `TPSAdvertisementsListeners`. These listeners then create the `TPSWireServiceFinder` (see Section 4.4.3 for a non-generic version of it) which is responsible for looking up the corresponding wire service and creating the input and output pipes (`TPSMyInputPipe` and `TPSMyOutputPipe` classes). When a user wants to subscribe to a specific type, he must provide a call-back object and an exception handler. These objects are saved by the `TPSSubscriberManager` which must also trigger the readers (`TPSPipeReader`) in order to receive the events.

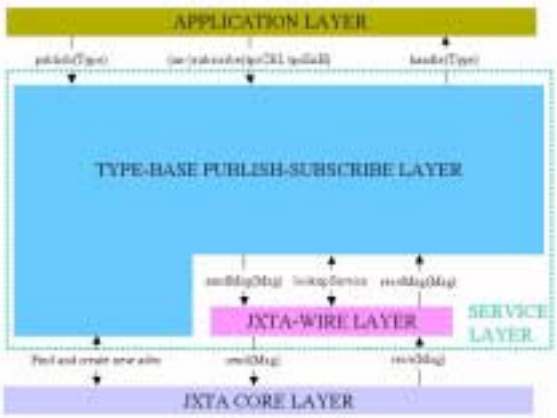

Fig. 9: General architecture

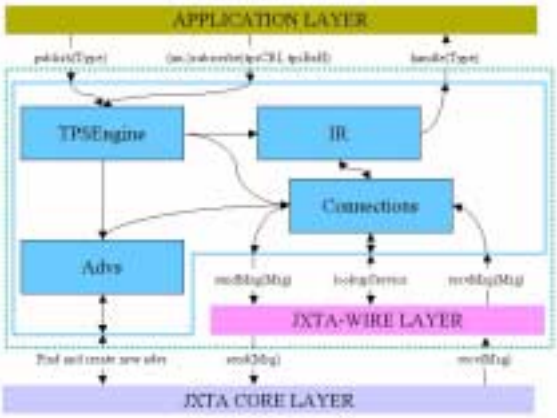

Fig. 10: General architecture (details of the TPS layer)

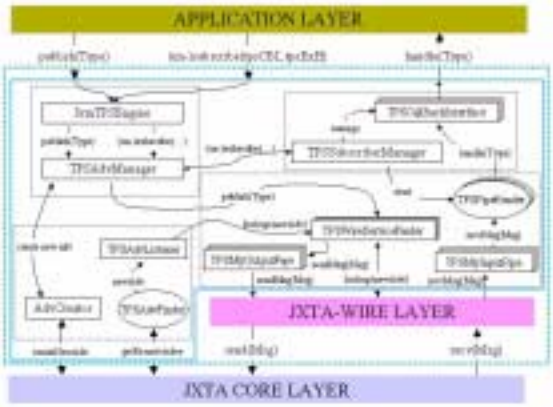

Fig. 11: General Architecture (details of the various blocks)

4. The programming experience

We compare here the programming of a typical P2P application using our TPS abstraction with the programming of the very same application using JXTA directly.

4.1 Ski-Rental Application

If you want to go skiing, you need skis. If you do not have any, you have two possibilities, either you buy them or you rent them. In the latter case, you will typically go to different shops in order to see what kind of skis you want and also to compare the different prices. Of course, nowadays, you could also do that online, by visiting differents web-sites. However, you must spend time doing that: you must stay behind your computer trying to find the best skis. A more comfortable way to do that is to use the TPS paradigm over a P2P infrastructure. You would then subscribe to the ski-rental type and wait for the answers. The infrastructure will be responsible for sending the subscription to the other peers and also getting the responses. You can now do something else during the search phase of the program and come back later to get the answers when they are available. Figure 12 and Figure 13 depict two GUIs that a publisher and a subscriber have with our ski-rental application.

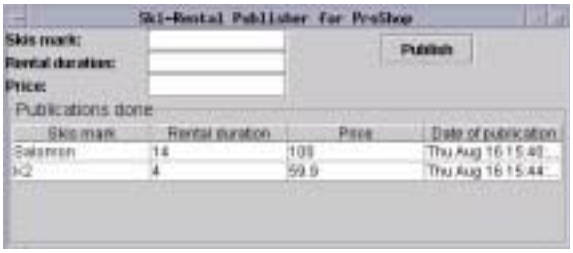

Fig. 12: GUI of a ski-rental publisher

Fig. 13: GUI of a ski-rental subscriber

When the publisher (a shop for example) starts, a search for a `SkiRental` advertisement is first launched. If the application does not find such advertisement in a specific amount of time, it creates its own one, but keeps trying to find others in order to send messages to the maximum number of interested subscribers. After that, an output pipe is created to send messages, and the window is displayed (see Figure 12).

In Figure 12, we can see that the shop seller can set the different options for the kinds of skis he wants to offer for rent (the brand, the duration of the rental, the price, ...). After setting these parameters, he just has to click on the publish button and the proposition is sent to all the interested subscribers via the output wire pipe.

For the subscriber, the same kind of initialization is done as for the publisher. Once this initialization has been accomplished, the window is displayed (see Figure 13). In this interface, the subscriber can see the different propositions from publishers of ski-rental advertisements. After some time, the subscriber can choose the best propositions and, maybe, send an e-mail to the shop.

Of course, these different GUIs are just examples. We can create more complicated ones using the complete set of methods that our implementation provides.

4.2 Programming Phases

Programming a TPS application, like the ski-rental, can be divided into four main phases, as depicted in Figure 14 (the arrows conveys the causality). In our implementation, we associate one instance of a publish/subscribe engine per type. If a publisher (or a subscriber) is interested in several “unrelated” types (i.e., different types that do not belong to the same type hierarchy), several instances of the publish/subscribe engine for each type of interest must be created. In Figure 14, this is conveyed by the fact that the type definition phase preceeds the initialization phase.

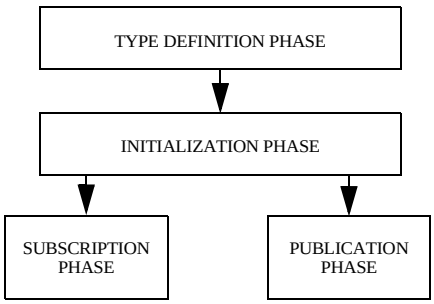

Fig. 14: The 4 different phases

In the following, we overview the two different ways of developing the ski-rental application according to the four phases, first using our TPS API and second using directly JXTA.

4.3 Renting skis with TPS

We present here the different phases shown in Figure 14 to create a simple application using our TPS architecture.

4.3.1 Type definition phase: the `SkiRental` type

We give here the basic type used in our application. This type contains the name of the renter, the price, the brand of the skis and the number of days the skis need to be rented. Here is the sketched source code of our simple type:

```
public class SkiRental implements Serializable {
   public SkiRental(String shop, float price, String brand, float numberOfDays) {...}
   public String toString() {...}
}
```

4.3.2 Initialization phase

When a user wants to use the TPS API, he must first write few lines to initiliaze the publish/subscribe engine:

```
TPSEngine<SkiRental> tpse = new TPSEngine<SkiRental>();
TPSInterface tpsInt = tpse.newInterface(“JXTA”, null, new SkiRental(), argv);
```

In the first line, we create the publish/subscribe engine and specify the type of interest. In the second line, the second parameter specifies a criteria we want for filtering advertisements (may be null). The third parameter is an instance of the type of the events we are interested in. We must provide this instance because GJ does not provide runtime information about (actual) type parameters. The last parameter denotes the arguments of the main class (may be null).

4.3.3 Subscription phase

To subscribe to events, one must create two objects (as described when presenting the `subscribe()` method, Figure 8): one implementing the `TPSCallBackInterface` interface and another one implementing the `TPSExceptionHandler` interface. Here is an implementation of the first interface:

```
public class MyCBInterface implements TPSCallBackInterface<SkiRental> {
   public void handle(SkiRental skiR) throws CallBackException {
     System.out.println("Skis that could be rented: "+skiR);
   }
}
```

This class defines what needs to be done when new events are received. In this case, we just print the events into the console. Here is a sketched implementation for the second interface:

```
public class MyExHandler implements TPSExceptionHandler<SkiRental> {
   public void handle(Throwable th) {...}
}
```

Besides these two classes, here are the lines one must add to subscribe to the type (`SkiRental`) specified before:

```
MyCBInterface mCBInt = new MyCBInterface();
MyExHandler mExH = new MyExHandler();
tpsInt.subscribe(mCBInt, mExH);
```

4.3.4 Publication phase

Up to now, we have only seen the subscriber perspective. If a publisher wants to publish an instance of the `SkiRental` type, here is the line he must add after the initialization phase:

```
tpsInt.publish(new SkiRental("XTremShop", "Salomon", 14f, 100f);
```

4.4 Renting skis with JXTA

Our aim here is to create the very same application than the one with TPS, i.e., an application with the same functionalities[1] as TPS. To provide these functionalities, we have re-created the same architecture presented in Figure 9, Figure 10 and Figure 11, but without the TPS flavour. We give here the sketched source code of those files described in Figure 11 (which are anyway required by a JXTA programmer to develop a classic JXTA-WIRE application). These files are the following: (1) `AdvertisementsCreator`, (2) `AdvertisementsFinder`, (3) `WireServiceFinder` [LPD01].

We will not develop the same structure as in Section 4.3, but we will present in detail these three files to convey our claim that TPS hides a lot of programming details. For example, writing the very same application with JXTA implies writing about 5000 lines of code[2] more than using directly TPS. Moreover, TPS allows the programmer to focus only on the portion of the code he is interested in.[3] This pre-

1. (1) Minimization of the number of advertisements for the same type, (2) management of multiple advertisements at the same time and (3) handling of duplicate messages.
2. If the user wants to use the full API. Otherwise (not having the functionnalities of TPS), the API saves, at least, to code 900 lines.
3. For presentation simplicity, we have used an example exploiting the static flavor of TPS (akin to subject-based publish/subscribe). Using the dynamic flavor of TPS lets the developper write even more code, which unlike when using JXTA directly, nevertheless remains concise.

vents from spending time learning the underlying JXTA concepts. Finally, TPS prevents the programmer from performing wrong type casts at runtime, and hence saves precious debugging time.

4.4.1 The AdvertisementsCreator class

One of the first things a new JXTA application must do, is to tell the world that there are new resources to share, i.e., to publish advertisements such as those presented in Section 2.1. In our Ski-Rental application, if the `AdvertisementsFinder` (see next section) does not find a `SkiRental` advertisement, the `AdvertisementsCreator` must create one in order to publish events and also for the other peers to be informed of this new interest. Figure 15 sketches the source code of the `AdvertisementsCreator` class.

```
1.    public class AdvertisementsCreator {
2.
3.       public AdvertisementsCreator(PeerGroup rootGroup, Discovery discoveryService) {
4.          this.rootGroup = rootGroup;
5.          this.discoveryService = discoveryService;
6.       }
7.
8.       public PeerGroupAdvertisement createPeerGroupAdvertisement(String name) throws Exception {
9.          String localPeerId = this.rootGroup.getPeerID().toString();
10.         PipeAdvertisement pipeAdv = (PipeAdvertisement)AdvertisementFactory.
11.                                                   newAdvertisement(PipeAdvertisement.getAdvertisementType());
12.         pipeAdv.setPipeID(new PipeID(this.rootGroup.getID()));
13.         pipeAdv.setName(name);
14.
15.         PeerGroup par = this.rootGroup;
16.         PeerGroupAdvertisement adv = (PeerGroupAdvertisement)AdvertisementFactory.
17.                                                 newAdvertisement(PeerGroupAdvertisement.getAdvertisementType());
18.
19.         adv.setPid(localPeerId);
20.         adv.setGid(new PeerGroupID().toString());
21.         adv.setName(PS_PREFIX + pipeAdv.getName());
22.         adv.setServiceAdvertisements(par.getAdvertisement().getServiceAdvertisements();
23.         adv.setApp(par.getAdvertisement().getApp());
24.         adv.setGroupImpl(par.getAdvertisement().getGroupImpl());
25.         Hashtable services = adv.getServiceAdvertisements();
26.
27.         ServiceAdvertisement wireAdv = new ServiceAdv();
28.         wireAdv.setName(WireService.WireName);
29.         wireAdv.setVersion(WireService.WireVersion);
30.         wireAdv.setUri(WireService.WireUri);
31.         wireAdv.setCode(WireService.WireCode);
32.         wireAdv.setSecurity(WireService.WireSecurity);
33.         wireAdv.setPipe(pipeAdv);
34.         wireAdv.setKeywords(pipeAdv.getName());
35.         adv.setIsRendezvous(true);
36.
37.         ServiceAdvertisement r = (ServiceAdvertisement)services.get ("jxta.service.resolver");
38.         Vector p = r.getParams();
39.         p.addElement(localPeerId);
40.         r.setParams(p);
41.         services.put("jxta.service.resolver", r);
42.
43.         services.put(WireService.WireName, wireAdv);
44.         adv.setServiceAdvertisements(services);
45.
46.         this.advertisement = adv;
47.         return adv;
48.      }
49.
50.      public void publishAdvertisement(Advertisement adv, int kindOfAdvertisement) {
51.         this.discoveryService.publish(adv, kindOfAdvertisement);
52.         this.discoveryService.remotePublish(adv, kindOfAdvertisement);
53.      }
54.}
```

Fig. 15: The `AdvertisementsCreator` class

As shown in the above code example, the creation of a `PeerGroupAdvertisement` in which we include the `WireService`, is not straightforward. A `PipeAdvertisement` needs to be created (lines 10-13), which is used by the `WireService` (line 33). After that, a `PeerGroupAdvertisement` needs to be created (lines 16-24) and, finally, the `WireService` is added to it (lines 27-44). Please note that, in our application, the name of the `PipeAdvertisement` (line 13) is the name of the type we are interested in.

Once the `PeerGroupAdvertisement` (containing the `WireService`) has been created, it is published in order for the other peers to find it and use it. This is done by the `publishAdvertisement()` method (lines 50-53). This method uses the JXTA `DiscoveryService` to send the advertisement to the other peers. We can distinguish two calls here. The first call writes the advertisement to the stable storage of the peer (if any), in order for the peers that are looking for advertisements to find that peer (line 51) . The second call sends the advertisements to the other peers via the standard used protocols (e.g, IP-Multicast, TCP or HTTP, line 52).

4.4.2 The AdvertisementsFinder class

```
1.public class AdvertisementsFinder implements Runnable, AdvertisementsFinderInterface, DiscoveryListener {
2.
3.       public AdvertisementsFinder(int typeOfAdvertisement, Discovery discoveryService, String prefix) {
4.          this.typeOfAdvertisement = typeOfAdvertisement;
5.          this.discoveryService = discoveryService;
6.          this.prefix = prefix;
7.       }
8.       public void run() {
9.          this.discoveryService.flushAdvertisements(null, Discovery.ADV);
10.         this.discoveryService.flushAdvertisements(null, Discovery.PEER);
11.         this.discoveryService.flushAdvertisements(null, Discovery.GROUP);
12.
13.         while (this.goOn) {
14.            switch (this.typeOfAdvertisement) {
15.               case Discovery.GROUP:
16.                  this.discoveryService.getRemoteAdvertisements(null, this.typeOfAdvertisement, "Name", this.prefix+"*",
17.                                                                this.NUMBER_OF_ADV_PER_PEER);
18.                  break;
19.            }
20.            Thread.currentThread().sleep(this.SLEEPING_TIME);
21.            Enumeration enum = null;
22.            switch (this.typeOfAdvertisement) {
23.               case Discovery.GROUP:
24.                  enum = this.discoveryService.getLocalAdvertisements(this.typeOfAdvertisement, "Name",
25.                                                                      this.prefix+"*");
26.                  break;
27.            }
28.            while (enum != null && enum.hasMoreElements()) {
29.               this.handleNewAdvertisement((Advertisement)enum.nextElement(), this.typeOfAdvertisement);
30.            }
31.         }
32.      }
33.
34.      private synchronized void addAdvertisement(Advertisement adv) {
35.         this.advertisements.add(adv);
36.         Enumeration enum = this.advertisementsListener.elements();
37.         while (enum != null && enum.hasMoreElements()) {
38.            ((AdvertisementsListenerInterface)enum.nextElement()).handleNewAdvertisements(adv);
39.         }
40.      }
41.
42.      public boolean findAdvertisement(Vector advVector, Advertisement adv) {
43.         try {
44.            Enumeration enum = advVector.elements();
45.            if (adv instanceof PeerGroupAdvertisement) {
46.               PeerGroupAdvertisement peerGAdv = (PeerGroupAdvertisement)adv;
47.               if (peerGAdv.getGid() != null) {
48.                  while (enum != null && enum.hasMoreElements()) {
49.                     PeerGroupAdvertisement element = (PeerGroupAdvertisement)(enum.nextElement());
50.                     if (peerGAdv.getGid().compareTo(element.getGid()) == 0) {
51.                        return true;
52.                     }
53.                  }
54.               }
55.            }
56.            else {return true;}
57.         }
58.         catch(ClassCastException cce) {cce.printStackTrace();}
59.         return false;
60.      }
61.
62.      private void handleNewAdvertisement(Advertisement adv, int typeOfAdvertisement) {
63.         switch (typeOfAdvertisement) {
64.            case(Discovery.GROUP) :
65.               if (adv instanceof PeerGroupAdvertisement) {
66.                  if (!this.findAdvertisement(this.advertisements, adv)) {
67.                     this.addAdvertisement((PeerGroupAdvertisement)adv);
68.                  }
69.               }
70.            break;
71.         }
72.      }
73.}
```

Fig. 16: The `AdvertisementsFinder` class

In this class, once an interesting advertisement is found, it is added to a vector of advertisements and dispatched to the registered `AdvertisementsListener` [LPD01]. The sketched source code of the `AdvertisementsFinder` (we give here only the part responsible for finding groups advertisements) is given in Figure 16.

Each instance of this class denotes a new Java thread which looks for new advertisements (in this special case, `PeerGroupAdvertisements`) until the application terminates. When the thread starts, old advertisements that could be in the peer cache are flushed (lines 9-11), and a remote query to search for new `PeerGroupAdvertisements` (lines 16-17) is sent. After a while, if new advertisements are received (lines 24-25), the `handleNewAdvertisement` method is invoked. In this case, the new advertisement is added to a vector and dispatched to the registered listeners (lines 34-40).

4.4.3 The WireServiceFinder class

```
1.public class WireServiceFinder {
2.
3.       public WireServiceFinder(PeerGroup peerGroup, PeerGroupAdvertisement pgAdv) {
4.          this.peerGroup = peerGroup;
5.          this.pgAdv = pgAdv;
6.       }
7.
8.       public Pipe lookupWireService() throws Exception {
9.          if (this.peerGroup != null && this.pgAdv != null) {
10.            this.wireGroup = PeerGroupFactory.newPeerGroup();
11.            this.wireGroup.init(this.peerGroup, this.pgAdv);
12.            this.pipeService = (Pipe)(this.wireGroup.lookupService(WireService.WireName));
13.            return this.pipeService;
14.         }
15.         throw new WireServiceFinderException("Unable to lookup the wire service");
16.      }
17.
18.      protected PipeAdvertisement getPipeAdvertisement(Pipe pipe) {
19.         ServiceAdvertisement sAdv = null;
20.         sAdv = (ServiceAdvertisement)pipe.getAdvertisement();
21.         if (sAdv == null) {
22.            return null;
23.         }
24.         return sAdv.getPipe();
25.      }
26.
27.      public MyInputPipe createInputPipe() throws WireServiceFinderException {
28.         PipeAdvertisement pAdv = this.getPipeAdvertisement(this.pipeService);
29.         try {
30.            this.myInputPipe = new MyInputPipe(this.pipeService.createInputPipe(pAdv),this.pgAdv);
31.            return this.myInputPipe;
32.         }
33.         catch (Exception e) {
34.            throw new WireServiceFinderException("Unable to create the input pipe.");
35.         }
36.      }
37.
38.      public MyOutputPipe createOutputPipe() throws WireServiceFinderException {
39.         PipeAdvertisement pAdv = this.getPipeAdvertisement(this.pipeService);
40.         try {
41.            this.myOutputPipe = new MyOutputPipe(this.pipeService.
42.                                    createOutputPipe(pAdv, Pipe.NonBlocking, TIME_TO_WAIT), this.pgAdv);
43.            return this.myOutputPipe;
44.         }
45.         catch (Exception e) {
46.            throw new WireServiceFinderException("Unable to create the output pipe.");
47.         }
48.      }
49.
50.      public void publish(Message msg) throws IOException {
51.         this.myOutputPipe.send(msg.dup());
52.      }
53.}
```

Fig. 17: The `WireServiceFinder` class

This class does three simple things: (1) searches for a `WireService` from a `PeerGroupAdvertisement`, (2) creates input and ouput pipes and (3) sends the events. The sketched source code is given in Figure 17.

A lookup of this `WireService` is triggered through the `lookupWireService()` method (lines 8-16). In this method, once the `WireService` is found, the Pipe object is retrieved from it (line 12). This object is used in the `createInputPipe()` and `createOutputPipe()` methods (lines 27-36 and lines 38-48). These methods just make a call to the appropriate methods of the `Pipe` class (we just embed here the result in a `MyInputPipe` or a `MyOutputPipe` object depending on the calls) after having extracted the `PipeAdvertisement` from the `Pipe` object (lines 18-25). To send messages to the newly created output pipe, a call to the `send()` method of the `MyOutputPipe` object is performed (which has the same signature than the standard JXTA `OutputPipe` class) (line 51).

5. The performance experience

This section presents the performance results of both our ski-rental application based on our TPS layer (SR-TPS), and implemented directly with JXTA (SR-JXTA). Even if JXTA-WIRE alone is not comparable with SR-TPS and SR-JXTA (since it does not insure the properties described in Section 4.4), we use it here as a (lower bound) reference point.

We used the following computer configurations: Sun Ultra 10 (CPU 440 MHz, RAM 256 MB) on Solaris 7; FastEthernet (100 Mbits/s); JXTA version 1.0 (build 30c, 08-24-2001); Java version "1.4.0-beta" (Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-beta-b65), Java HotSpot(TM) Client VM (build 1.4.0-beta-b65, mixed mode)); messages size: 1910 bytes.

We give here the invocation time[1] and throughput for a limited[2] number of participants. These tests aim at giving a hint about the differences between the three implementations.

5.1 Invocation time

We measured the time taken for calling the `sendMessage()` method: The publisher produces here 50 events one after. The results are given in Figure 18. We can see that the results are not linear at all. In fact, the one for JXTA-WIRE has a very big standard deviation (e.g., ~20% for a single subscriber and ~30% for four subscribers). Not surprisingly, JXTA-WIRE alone is quicker than SR-JXTA and SR-TPS. Note however that there is virtually no difference between SR-TPS and SR-JXTA (e.g., about 1% with one subscriber). The number of subscribers clearly affects the results. This can be explained by the fact that the more subscribers are involved, the more connections the publisher must handle and, consequently, the slower the invocation time is.

1. Since JXTA is not reliable (August 2001 release) and since we do not want to modify the JXTA implementation, we were not able to measure the latency. We focused on the invocation time instead.
2. At the time of our implementation, JXTA was not able to handle connections between more that 5 peers sending a lot of messages.

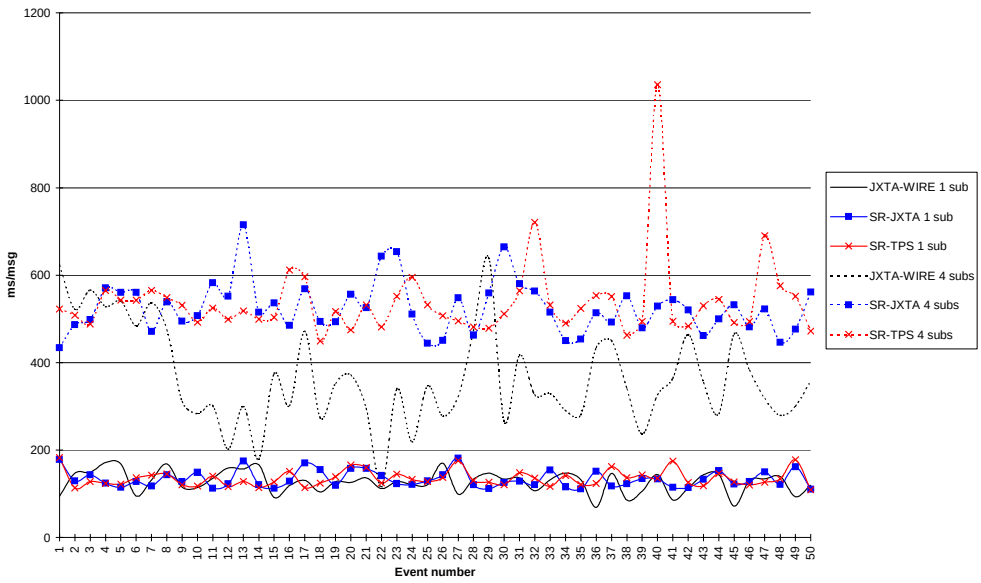

Fig. 18: Invocation time

5.2 Throughput: the publisher viewpoint

We consider here a set of 100 published events and we measure the time for the publisher to deliver those events to the subscriber(s). Again, the values for SR-JXTA and SR-TPS are very close. We can also notice that our different layers are slightly slower than JXTA-WIRE itself (e.g., about two events per second for one subscriber) (Figure 19). When the number of subscribers increases, the differences between the layers become insignificant (e.g., with four connected subscribers, only 0.3 events per second between JXTA-WIRE and SR-JXTA and 0.5 events per second between JXTA-WIRE and SR-TPS).

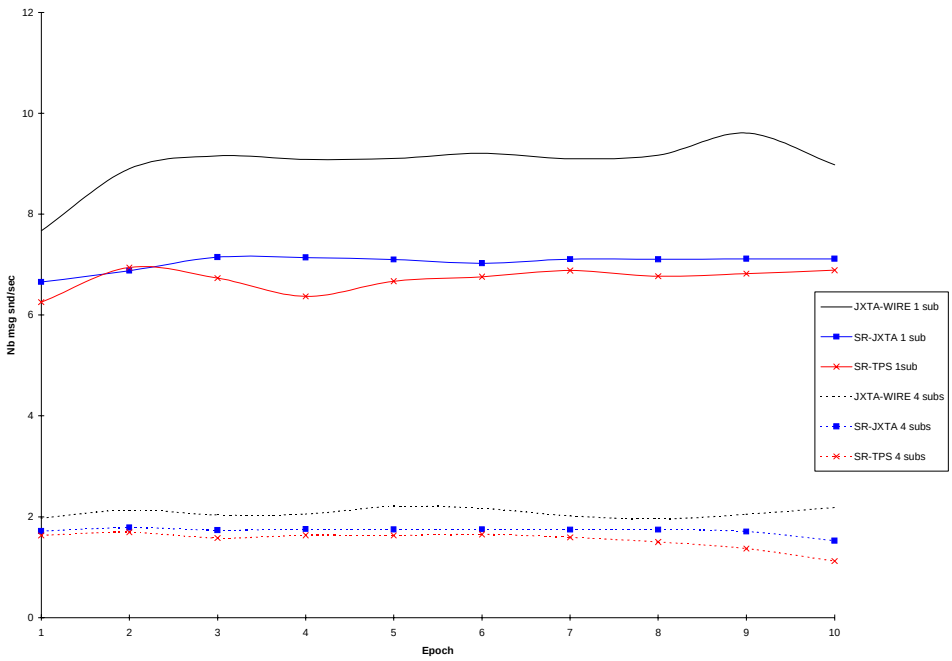

Fig. 19: Publisher's throughput

5.3 Throughput: the subscriber viewpoint

Here the publishers try to flood the subscriber (10000 events published per each publisher). Every second, we measure the number of events that are received; during 50 seconds. The results are given in Figure 20. Once again, we have a quite big standard deviation and the number of events received per second is not really stable. For example, with a single publisher, the average throughput for JXTA-WIRE is about 7.8 events per second and, for SR-JXTA and SR-TPS, the values are 6.1 and 6.0 respectively. If we compare these results with the ones from Figure 19, we can see that, for one publisher, JXTA-WIRE saturates. JXTA-WIRE can simply not handle all published events (e.g., about nine per second for one subscriber (see Figure 19)).

When we increase the number of publishers, the average number of events received per second remains quite the same for the different layers. Again, the average drops by a factor of about three (for the same reason explained in Section 5.1).

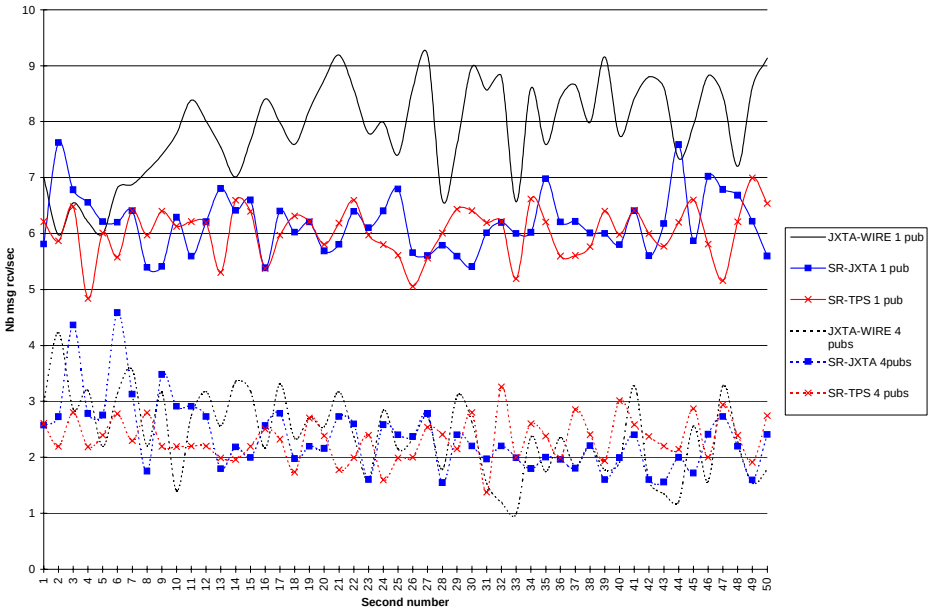

Fig. 20: Subscriber's throughput

6. Concluding Remarks

This experience paper makes a case for TPS (Type-Based Publish-Subscribe) as a viable alternative abstraction to RPC for future Internet-wide operating systems to support P2P applications. TPS fits particularly well the decoupled nature of server-less P2P applications. In short, TPS is simple to use (almost as simple as RPC), ensures type-safety and encapsulation (just like RPC) and yet preserves the decoupled nature of P2P applications (unlike RPC). This paper describes a TPS API and an implementation of TPS over JXTA, and then compares the programming and performance of a testbed application over TPS and directly over JXTA. Roughly speaking, (1) TPS makes the programming of a P2P application signifi-

cantly easier than using directly a library like JXTA and (2) does not introduce a significant overhead with respect to JXTA.

Our current TPS prototype is based on the JXTA release of August 24, 2001. New implementations of JXTA will obviously impact our prototype but we do not believe they will fundamentally impact the nature of the results drawn from the present experience.

Of course, more programming and implementation testbeds need to be performed before TPS can be realistically viewed as a reasonable general abstraction for P2P applications. In particular, measuring the lack of programming flexibility that our abstraction involves is not clear. We can for example easily see through our ski-rental application that our TPS API does not enable a subscriber to immediately reply to a publisher that posted an interesting event. This would require a combination with a more traditional RPC kind of interaction or directly using the underlying P2P library. Another loss of flexibility is our assumption that the different peers must a priori agree on the Java type system which is not the case when using JXTA directly. Figuring out "loose" ways of achieving such common knowledge at run-time (e.g., by representing types through XML data structures) is the subject of ongoing investigations.

References

[BN83] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'83)*. October 1983.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire and Ph. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 183-200. October 1998.

[CSW+00] I. Clarke, O. Sandberg, B. Wiley et al. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the International Computer Scientist Institute Workshop on Design Issues in Anonymity and Unobservability (ICSI'2000)*. July 2000.

[CRW00] A. Carzaniga, D. S. Rosenblum and A. L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*. July 2000.

[EGD01] P. Th. Eugster, R. Guerraoui and Ch. H. Damm. On objects and events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*. October 2001.

[EGS00] P. Th. Eugster, R. Guerraoui and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*. January 2000.

[GKK01] S. Grant, M. P. Kovacs, M. Kunnumpurath, et al. *Professional JMS*. Wrox Press, March 2001.

[Heim01] D. Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC'2001)*, pages 176-181. 2001.

[Gryp01] *Gryphon: Publish/Subscribe over public networks*. IBM T.J. Watson Research Center. http://researchweb.watson.ibm.com/gryphon/Gryphon/gryphon.html. February 2001.

[Moj01] J. Mojica. *Developing COM+ Servers with COM, COM+ and .NET*. Chapter 11 of the COM+ Programming with Visual Basic book. June 2001.

[Mon01] A. Montresor. Anthill: a Framework for the Design and the Analysis of Peer-to-Peer Systems. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01)*. May 2001.

[MS01] D. Malkhi and Y. Sella. Replication by Diffusion in Large Networks. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01)*. May 2001.

[MSO+01] N. Minar, C. Shirky, T. O'Reilly et al. *Peer-to-Peer Harnessing the Power of Disruptive Technologies*. O'Reilly. March 2001.

[Ore01] O'Reilly web site on p2p. http://www.openp2p.com. 2001.

[OPS+93] B. Oki, M. Pfluegl, A. Siegel et al. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*. December 1993.

[PFJ+01] J. Pereira, F. Fabret, H.-A. Jacobsen et al. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of the 4th ACM Special Interest Group on Management Of Data (SIGMOD'01)*. 2001.

[PFL+00] J. Pereira, F. Fabret, F. Llirbat et al. Publish/Subscribe on the Web at Extreme Speed. In *Proceedings of the 3rd ACM Special Interest Group on Management Of Data (SIGMOD'00)*. 2000.

[SUN01] Project JXTA web site. http://www.jxta.org. Sun Microsystems. 2001.

[LPD01] LPD web site. http://lpdwww.epfl.ch. For the full source codes, follow People / Sebastien Baehni / Current work.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, third edition. January 1996.

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

[Wea01] M. Weatherford. *P2P Acclaimed by Jury of Peers*. IEEE Distributed Systems Online. http://www.computer.org/dsonline. Volume 2, Number 3. 2001.

[WS98] D. J. Watts and S. H. Strogatz. *Collective dynamics of "small-world" networks*. Nature, vol. 393. June 1998.