TYPE-BASED PUBLISH/SUBSCRIBE

THÈSE N° 2503 (2001)

PRÉSENTÉE AU DÉPARTEMENT DE SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Patrick Thomas EUGSTER

Ingénieur informaticien diplômé EPF

originaire de Oberegg (AI)

Lausanne, EPFL
2001

Abstract

As the scale of communication networks becomes increasingly imposing, more and more applications aim at exploiting this high potential in connectivity. As a consequence, more sophisticated tools are required to make the development and deployment of such large scale applications more effective.

A fair amount of research has already been accomplished in the field of distributed programming, however, with the continuously increasing number of network nodes hiding behind the vague notion of "large scale", there is still a high demand for (1) new programming abstractions, and obviously also (2) algorithms implementing these abstractions in a way that does not diminish their scalability properties.

In the case of abstractions, the ever prominent *remote procedure call* (RPC) abstraction for distributed programming has celebrated a tremendous success for its hiding of distribution to a large extent, yet has turned out to be inadequate for large scale applications: initially conceived for pairwise interaction between client/server application components, the RPC becomes increasingly inefficient in *one-to-n* interactions underlying most large scale applications as the very n grows.

The *publish/subscribe* paradigm has been proposed as a candidate distributed programming abstraction to meet the scalability requirements of todays applications. Unfortunately, existing implementations promote low-level programming models. The conveyed data is viewed as primitive structures, requiring explicit transformation of higher-level data abstractions. These transformations are error-prone, due to a lack of type safety, and provide no encapsulation of the exchanged data.

Deterministic dissemination algorithms for asynchronous distributed systems, such as the Internet, have similarly been shown to inherently scale poorly. As an alternative, probabilistic, *gossip-based*, algorithms have been devoted much attention recently. Existing algorithms however do not deal with the filtering of data aimed at satisfying every component's precise interests in a strongly dynamic setting.

This thesis presents the *type-based publish/subscribe* (TPS) paradigm, and an algorithm to implement it. TPS is to publish/subscribe what RPC is to synchronous message passing: a higher-level paradigm variant, which hides "ugly" aspects of distribution and fits naturally into an object-based programming model. We illustrate TPS through Java, first by implementing it as a library, and second, through specifically added language primitives. We propose an original gossip-based multicast algorithm, which allows the exploitation of the scalability offered by TPS at the abstraction level.

Résumé

L'échelle des réseaux de communications devient de plus en plus importante. De ce fait, un nombre croissant d'applications tente d'exploiter ce grand potentiel en connectivité. Par conséquent, des outils plus sophistiqués sont nécessaires afin de rendre plus efficace le développement et déploiement d'applications à large échelle.

Un effort de recherche considérable à déjà été accompli dans le domaine de la programmation répartie. Avec le nombre croissant de noeux de réseaux se cachant derrière la notion vague de "large echelle", une grande demande existe néanmoins encore pour des (1) abstractions de programmation adéquates, et évidemment aussi pour des (2) algorithmes implémententant ces abstractions d'une manière qui ne diminue par leur extensibilité.

Dans le cas des abstractions, le fameux *appel de procédure à distance* utilisé dans la programmation répartie a été un succès faramineux grâce à sa capacité de cacher la distribution dans une certaine mesure, mais s'est avéré insuffisant pour des applications à large échelle: initialement con cu pour une interaction bilaterale entre des composants clients/serveurs, l'appel de procédure à distance devient de moins en moins efficace dans des interactions *un-à-n* sousjacentes pour la plupart d'applications dites large échelle, lors que n croit.

Le paradigme de *publication/souscription* a été proposé comme abstraction possible pour la programmation répartie, afin de satisfaire les besoins en termes d'extensibilité des applications actuelles. Malheureusement, les implémentations existantes promouvoient des modèles de programmation de bas niveau. Les données échangées sont vues comme des structures primitives demandant des transformations explicites d'abstractions de données de plus haut niveau. Ces transformations sont soumises à des erreurs reflètant un manque de sûreté du typage, et n'offrent aucune encapsulation des données transférées.

Similairement, il a été démontré que des algorithmes de dissemination déterministes pour des systèmes reparties asynchrones, tel qu'Internet, sont faiblement extensibles. Comme alternative, beaucoup d'attention a été récemment consacrée aux algorithmes *probabilistes.* Cependant, les algorithmes existants ne gèrent pas le filtrage des données nécessaire pour satisfaire les intérêts précis de chaque composant dans un environnement hautement dynamique.

Cette thèse présente le paradigme de *publication/souscription basé sur les types* (TPS), et un algorithme servant a implémenter ce paradigme. TPS représente pour le paradigme de publication/souscription "classique", ce que l'appel de procédure à distance représente pour le passage de messages synchrone: une variante de paradigme de plus haut niveau, qui cache certains aspects indésirés de la distribution, et qui ainsi se marie bien avec un modèle de programmation par objets. Nous illustrons TPS par Java, d'abord en l'implémentant comme une librarie, et ensuite, en ajoutant des primitives de langage spécifiques. Nous proposons un nouvel algorithme de multicast probabiliste, qui permet d'exploiter au mieux le potentiel d'adaptation à large echelle offert par TPS en tant qu'abstraction.

To the Inventors of Alka-Seltzer ®...

Acknowledgements

Above all, I would like to express all my gratitude to my supervisor, Prof. Rachid Guerraoui, for combining, at perfection, the role of supervisor with those of colleague, and also friend. His constant support, both in terms of encouragement as well as ideas, have given the best possible foundation for this work.

I am eternally grateful also to my "fellow-sufferer" and friend Romain Boichat for his constant support and advice. Such good friends are hard to find.

My thanks go also to the members of the jury, for the time they have spent applying their expert knowledge to the examination of this thesis: Prof. Klaus R. Dittrich, Prof. Ole Lehrmann Madsen, Prof. Martin Odersky, Dr Robbert van Renesse, and Dr Joseph Sventek. Also, I am very grateful to Prof. Emre Telatar for accepting to head my PhD examination.

Prof. Martin Odersky (and his team), and Dr Joseph Sventek deserve a special mention, as, just like Nicolas Ricci, they have considerably contributed to the accomplishment of this thesis through extremely valuable input and feedback.

I am also thankful to all the other people I was brought to work with. Among these, I would like to thank Dr Pascal Felber, Dr Benoît Garbinato, and other members of the "former lab" for fruitful collaboration and good times. Very special thanks go to Christian Heide Damm, with whom I have especially enjoyed working, and all "current lab" members I have had the pleasure of collaborating with, Petr Kouznetsov, Sidath Handurukande, and Sébastien Baehni. Of course many thanks go to the lab's good spirit, Kristine Verhamme, for taking care of so many important issues.

Last but not least, I would like to thank all my beloved; my whole family and Sandra for their love and support, and of course, all my friends :-)

Preface

During my time as Ph.D. student and research assistant, I was brought to work around different subjects. Before starting my actual Ph.D. studies, I was involved in research around object group systems, including technologies such as CORBA or DCOM, and around underlying group communication issues. I have co-authored two publications in that area [FDES99,GEF$^+$00].

In the context of this thesis, I have, on the one hand, worked on abstractions for distributed programming, and, on the other hand, on algorithms implementing these abstractions.

The former axis of research has resulted in various publications around the subject of publish/subscribe [EGS00,EG01a,EGS01,EGD01,EG00,EFGK01,DEG01], of which mainly the first four are covered by this thesis. I am currently involved in the writing of a book about distributed programming abstractions in general. The second research direction was concerned with dissemination algorithms for publish/subscribe systems. I have co-authored several publications related to this subject, mainly on probabilistic algorithms [EBGS01,EGH$^+$01,EKG01,EG01b]. The algorithm related by the last paper is included in this thesis as well.

[FDES99] P. Felber, X. Défago, P.Th. Eugster, and A. Schiper. Replicating CORBA Objects: A Marriage between Active and Passive Replication. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS '99)*, pages 375–387, June 1999.

[GEF$^+$00] R. Guerraoui, P.Th. Eugster, P. Felber, B. Garbinato, and K. Mazouni. Experiences with Object Group Systems. *Software - Practice and Experience*, 30(12):1375–1404, October 2000.

[EGS00] P.Th. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[EG01a] P.Th. Eugster and R. Guerraoui. Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 131–146, January 2001. Awarded *best student paper.*

[EGS01] P. Th. Eugster, R. Guerraoui, and J. Sventek. Loosely Coupled Components. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, chapter 8. Kluwer, 2001.

[EGD01] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

[EG00] P.Th. Eugster and R. Guerraoui. Type-Based Publish/Sub-scribe. Technical Report DSC/2000/029, Swiss Federal Institute of Technology, Lausanne, June 2000. In submission.

[EFGK01] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. Technical Report DSC/2001/004, Swiss Federal Institute of Technology, Lausanne, January 2001. In submission.

[DEG01] C.H. Damm, P.Th. Eugster, and R. Guerraoui. Abstractions for Distributed Interaction: Guests or Relatives?. Technical Report DSC/2001/052, Swiss Federal Institute of Technology, Lausanne, September 2001. In submission.

[EBGS01] P.Th. Eugster, R. Boichat, R. Guerraoui, and J. Sventek. Effective Multicast Programming in Large Scale Distributed Systems. *Concurrency and Computation: Practice and Experience*, 13(6):421–447, May 2001.

[EGH+01] P.Th. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight Probabilistic Broadcast. In *IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, pages 443–452, June 2001.

[EKG01] P.Th. Eugster, P. Kouznetsov, and R. Guerraoui. $\Delta-$Reliable Broadcast. Technical Report DSC/2001/010, Swiss Federal Institute of Technology, Lausanne, January 2001. In submission.

[EG01b] P.Th. Eugster and R. Guerraoui. Probabilistic Multicast. Technical Report DSC/2001/051, Swiss Federal Institute of Technology, Lausanne, September 2001. In submission.

Contents

Chapter 1

Introduction

Large Scale Computing: Sometimes Size Does Matter

Large scale networks, like the Internet, or more recently, mobile and ad-hoc networks, represent an important step towards global connectivity. Exploiting these large infrastructures to provide useful services to its end-users is a primary concern for a vast industrial branch.

Devising software applications for these large scale networks in a way such that they meet the requirements of end-users both in simplicity but also in efficiency is very challenging.

Mastering the inherent complexity of distributed applications, i.e., the development of such applications in an acceptable time frame, is a non-trivial task, which has led to a proliferation of both research and industrial development around the subject of distributed programming.

Object-Oriented Programming: When One Small Step Becomes a Giant Leap

One very important advancement towards a rapid conception of such applications, indeed any application, was given by the powerful *object* abstraction. The resulting *object-oriented programming* led to a better *modularity*, *extensibility* and *reusability* of application components, reducing the burden on developers [Weg90].

The object paradigm has been straightforwardly applied to distributed settings, by leveraging a form of *remote procedure call* (RPC) [BN84] on objects: objects are distributed over a set of nodes, and functions are invoked the same way on remote objects than on local ones. The RPC has become very popular as a simple yet powerful abstraction for distributed object-oriented programming, mainly due to the fact that it provides *type safety* and *encapsulation*, by concealing details concerning distribution behind application-defined types.

The Limits of Engineering: One Size Can not Fit All

It has long been argued that distribution is an implementation issue and that the very well known metaphor of objects as "autonomous entities communicating via RPC" can directly represent the interacting entities of a distributed system. This approach has been conducted by the legitimate desire to provide distribution transparency, i.e., hiding all aspects related to distribution under traditional centralized constructs. One could then reuse, in a distributed context, a centralized program that was designed and implemented without distribution in mind.

As widely recognized however (e.g., [Gue99]), distribution transparency is a myth that is both misleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable with that of local interactions. The possibility of partial failures can fundamentally change the semantics of an invocation.

The RPC abstraction, initially conceived for strict pairwise interactions between objects, has been indeed adapted to reflect the unpredictability of the underlying network to some extent, but does not apply well to the dissemination of information to a wide audience of objects. This limited scalability manifests quickly by leading to unbearably poor performances as the audience targeted by an interaction grows in size. Yet, such *multicast*-style interactions are the very base for large scale applications. In 1998, a study [LAJ98] measured that 30% of Internet traffic consisted of multicast interactions, and already announced the growth of this proportion to over 50% at this day.

To meet the requirements of large scale distributed programming, several abstractions have been proposed or adapted, like *multicast sockets*, or the *shared space* [Gel85] abstraction widely employed in parallel and concurrent programming, which has been applied to object settings. These have however altogether failed in providing an appropriate high-level abstraction addressing both ease-of-use, and even more important, the stringent scalability requirements of modern applications.

Publish/Subscribe vs Objects: Out of Sight, Out of Mind

More recently, the *publish/subscribe* paradigm [OPSS93] has proven to be an adequate abstraction for large scale information dissemination. At that time, the desire for scalability and other properties like *platform interoperability* had become so strong, that some of the nice principles which had led to the success of the object paradigm were devoted little to no attention, and the market was invaded by publish/subscribe systems based on very pragmatic programming models.

In 1999, while observing the rapid proliferation of publish/subscribe engines and the promoted schemes, Koenig [Koe99] interprets the lacks for type safety and encapsulation in existing systems as an inherent impossibility of combining these principles with the publish/subscribe paradigm.

Back to Objects: When the Dust Settles

The truth is that, while other abstractions for distributed programming have benefited from thorough studies in the context of object-orientation, only little consideration has been given the publish/subscribe paradigm in the object community.

This thesis argues that publish/subscribe, until today, still remains the distributed programming abstraction offering the biggest potential for scalability at the abstraction level [EFGK01], and that it is very appealing to merge the benefits of objects with those of an asynchronous *event-based* programming model for distributed interactions such as publish/subscribe [EGS00]. We make a case for such a marriage, by describing a novel variant of the publish/subscribe paradigm, called *type-based publish/subscribe*, or in short, TPS, which serves as vehicle for smoothly integrating objects and events.

TPS aims at providing publish/subscribe those properties that made the RPC famous in object-oriented settings, mainly type safety and encapsulation with application-defined types, without hampering with the inherent scalability properties of publish/subscribe. We illustrate the feasibility of our concepts by instrumenting a statically typed object-oriented language, the ever popular Java^{TM}, with TPS; first, by putting TPS to work through a library based on GJ, an extension of Java [BOSW98], and second, by adding language primitives for TPS.

A Multicast Algorithm for TPS: Going the Whole Nine Yards

The best abstraction is worthless without a powerful underlying algorithm. While seeking for adequate multicast algorithms to implement TPS, we have devoted much effort to a rigorous study of *gossip-based algorithms* [DGH$^+$87]. In fact, these non-deterministic algorithms date back to the original USENET news protocol developed in early 1980's, and have experienced a true revival through the strong lack for scalable algorithms, with applications in various contexts (e.g., [vRMH98, vR00]). Gossip-based multicast algorithms fill this gap by providing very good practical reliability, and excellent scalability properties.

To illustrate how TPS preserves the scalable nature of publish/subscribe, we have developed a novel algorithm, custom-tailored to a strongly dynamic environment such as TPS, but nevertheless general enough to be applied in a wide set of contexts.

Our Hierarchical Probabilistic Multicast (hpmcast) algorithm relies on a hierarchical membership. This hierarchical orchestration of participants adds a notion of memory resource consumption scalability, by reducing the membership view of every participant, to the network resource consumption scalability inherent to gossip-based algorithms. Furthermore, we illustrate how this hierarchy enables the exploitation of the topology of the system, and at the same time, also commonalities between interests of participants in that system.

Roadmap: The Walk to the Talk

Chapter 2 overviews abstractions and paradigms for distributed programming. The chosen perspective is oriented towards illustrating the supremacy of the general publish/subscribe paradigm in terms of adequacy for scalable mass information dissemination. Also, we overview the evolution of prevalent abstractions for distributed programming in the face of object-oriented programming.

Chapter 3 discusses type-based publish/subscribe from an abstract point of view. We start by pinpointing the flaws of current publish/subscribe schemes with respect to object-orientation, and summarizing the rationale underlying the efforts reported in this dissertation.

Chapter 4 presents a seminal implementation of TPS based on a library approach. This library promotes a programming abstraction called Generic Distributed Asynchronous Collection (GDAC), which is implemented in Generic Java (GJ), a superset of the Java language encompassing genericity. We give some results on the cost of filtering events based on their methods, and we depict some optimizations.

Chapter 5 illustrates Java$_{PS}$, a home-grown extension of the Java language incorporating TPS. We relate the native Java language features which we have exploited, and detail the primitives and mechanisms added to inherently support TPS.

Chapter 6 presents *hpmcast*, our novel gossip-based multicast algorithm. We show how the hierarchical membership underlying *hpmcast* enables the reduction of the membership views of participants, as well as the exploitation of both locality of participants and redundancies in their individual interests.

We conclude this dissertation with remarks on a selected set of issues, including a comparison of our library and language integration approaches, and future efforts and application contexts for TPS.

Chapter 2

Background: Abstractions for Distributed Programming

Several paradigms for programming at a distributed scale appear in the literature. The most prominent paradigms and abstractions in this context include *message passing*, *remote procedure call*, *tuple spaces*, and *publish/subscribe*. We outline these interaction styles and the "typical" abstractions they rely upon.

Each of these abstractions has its respective advantages over others, making it preferable in some contexts. We discuss these interaction paradigms here with particular focus on their ability of providing *scalability* at the abstraction level. Akin to the classification introduced in [Gel85], we discuss this ability in terms of *coupling* of data producers and consumers, based on the assumption that reducing this coupling between producers and consumers reduces the dependencies in a distributed system, which in turn positively influences scalability. We discuss the case of publish/subscribe more in detail, conveying its advantages in terms of decoupling of participants.

2.1 Message Passing

Tightly coupled to the *socket* abstraction, message passing can be viewed as the most basic paradigm for distributed interaction. Though more elaborate interfaces exist for message passing, like the *Message Passing Interface* (MPI) [For98], we focus here on the socket abstraction as the best known abstraction for directly reflecting the nature of the underlying network.

2.1.1 Sockets

The socket abstraction represents a communication endpoint with respect to a low-level communication layer. Typical representatives are the layers 3 and 4 of the OSI stack, corresponding to the *network* and *transport* layers respectively [Tan96].

Primitives

Message passing is asynchronous for the producer, and is usually implemented by a `send()` primitive, which is invoked by passing the message as argument. Message consumption on the other hand is generally synchronous, and is triggered by invoking a `receive()` primitive which blocks until a message has been received and can be delivered to the consumer.

Language Integration

Like many "distributed" interaction paradigms, message passing has also extensively been used first in parallel and concurrent programming. Early, mainly procedural and process-oriented languages, like Occam [Pou84] (promoting named unidirectional connections) or also SR [AOC$^+$88] (first with *semi-synchronous* sends, later also with synchronous calls), inherently included some form of message passing, by providing primitives like `send()` and `receive()` as built-in procedures.

Coupling

As summarized in Table 2.1 at the end of this chapter, the producer and the consumer are coupled in *time*, meaning that they must be active at the same time. Parties are also coupled in *space*, that is, the destination of a message is explicitly specified. In other terms, the producer has an identifier or address referencing the consumer. Finally, consumers are coupled in *flow*, since the flow of control of consumers invoking the `receive()` primitive is blocked until a message is received and delivered. This flow coupling however can become more important, for instance in the case of *connected* sockets.

2.1.2 Connected Sockets

Connected sockets are tied to a remote address. This remote address is systematically used as a destination for outgoing messages, and the corresponding socket hence represents a connection with a remote party. For receiving messages, a socket is usually tied to its local address.

Channels and Streams

In practice, such sockets represent higher-level communication layers which offer connection-oriented interaction, e.g., *Transport Control Protocol* (TCP), *Sequenced Packet Exchange* (SPX) and consequently, give rather strong guarantees, like *reliable* delivery (*exactly-once*). Connected sockets often give the illusion of a dedicated network *channel* connecting the endpoints, and provide the application with a *stream* abstraction, on which data is written and read in the same order. As a consequence, the flow coupling with connected sockets is particularly strong, since the threads of execution of both producers and consumers are inherently synchronized: once invoked by a producer, a `send()` primitive only returns after the message has been received by the consumer, as shown in Figure 2.1.

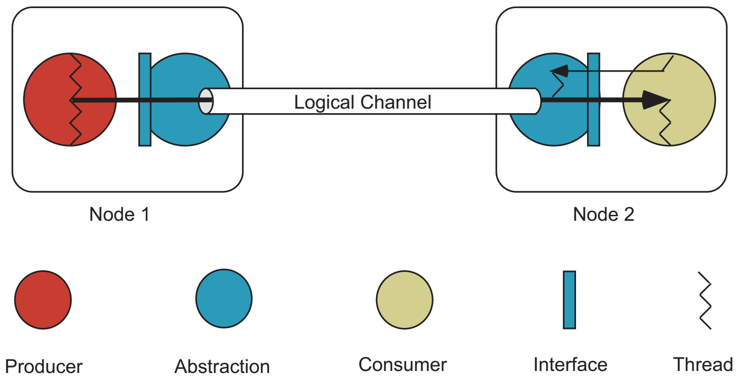


Figure 2.1: Message Passing Interaction with a Connected Socket

Anthropomorphism: A Phone Call Away

For all abstractions presented in this chapter, we will attempt intuitive illustrations of their characteristics through anthropomorphisms.

Communication based on a connected socket, for instance, can be roughly compared to a telephone call. The two involved parties have to be there at the same time, and the initiator at least has to “know” the callee by its phone number. Once established, communication is bidirectional. Participants are however very aware of the fact that they are not directly interacting, but merely communicating via an intermediate mechanism.

2.1.3 Unconnected Sockets

In contrast to connected sockets, *unconnected* sockets are not tied to any remote address, and can be used to send to virtually any address. The destination address is hence given as a parameter every time the `send()` primitive is invoked, and there is no illusion of a reserved network channel (cf. Figure 2.2).

Datagrams

In practice, as shown by the *User Defined Protocol* (UDP) or the *Internetwork Package Exchange* (IPX), unconnected sockets represent rather low-level communication layers which do not have any notion of flow control. Producers send *datagrams* (single packets) one-by-one to consumers, benefiting only from weaker semantics than in the case of connected sockets (e.g., *best-effort*, *at-most-once*). In contrast, these weaker guarantees offered by unconnected sockets are easier to implement and can be satisfied without blocking producers.

Flow coupling is thus slightly reduced, since the threads of a message producer and its consumer are not synchronized, and one could even imagine to view time decoupling as relaxed; a consumer could be unavailable at the moment at which a message is sent to it, but could become available while the message is in transit. This remark can be extended to many interaction paradigms, but we do not consider such "glitches" here.

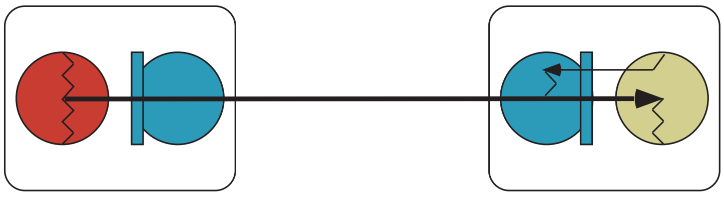


Figure 2.2: Message Passing Interaction with an Unconnected Socket

Anthropomorphism: Short Message Service (SMS)

In the case of an unconnected socket, one can draw a comparison with sending messages via *short message service* (SMS) from a mobile phone, a communication mode which has become increasingly popular in the GSM mobile world. Indeed, the sender edits a message, before choosing the destination phone number. On the other hand, if the receiver needs the transmitted information, we will have to wait for the message to arrive.

One could also compare unconnected sockets with answering machines. Instead of trying to call a same person several times, until that person is finally reached, one can simply leave a message on the answering machine. The guarantees provided are however rather weak, since the message can be ignored by the targeted person, or can be dropped by older models of answering machines whose memory capacities are exceeded.

2.1.4 Multicast Sockets

The "classic" sockets represent endpoints for one single participant, and the resulting interaction is sometimes classified as *one-to-one*, since it inherently involves two remote parties. In contrast, the term *one-to-n* (*one-to-many*) is used to denote sockets which are characterized by logical addresses capturing several effective parties (or any other such abstractions). *Network-level protocols* like *UDP Broadcast* or *IP Multicast* [Dee91], or more sophisticated protocols (cf. Section 6) are usually represented by multicast sockets.

Connected Vs Unconnected

The choice between connected and unconnected sockets can be considered orthogonal to the *dissemination model* (one-to-one vs one-to-n). As long as low-level protocols are involved, it makes sense to provide unconnected interaction, while stronger guarantees could be implemented with connected sockets, even if several remote parties are involved.

Multicast vs Broadcast

Note in this context that in the literature the terms *multicast* and *broadcast* are sometimes used interchangeably, and sometimes separated according to various criteria.

Besides in names of cited products, protocols or algorithms, the term *broadcast* will be used in this thesis to denote a one-to-n dissemination scheme with more than one potential destination. At every single primitive interaction, the *entire* set of potential destinations is invariably addressed.

The term *multicast* will be used to refer to a one-to-n interaction with *some-of-n* flavor. Precisely, this specific type of one-to-n interaction potentially involves more than one destination just like a broadcast, but the effective set of destinations is in general merely a subset of the potential set of destinations, and the former set can differ from one simple interaction to another. In contrast, broadcast can hence be viewed as manifesting a one-to-n dissemination model with *all-of-n* flavor.

Anthropomorphism: Group SMS

Several mobile communication companies already provide the possibility of sending short messages atomically to a set of phone numbers. While most mobile phones allow the explicit definition of several destinations when sending such messages, these mobile communication services include the transparent management of "group" numbers.

2.2 Remote Procedure Call (RPC)

In any case, sockets represent an essential, yet low-level abstraction, on top of which fewer distributed applications are nowadays built from scratch, since network addresses, the transformation of data to a more "physical" representation, and sometimes also flow control (e.g., retransmissions), have to be dealt with explicitly.

A more high-level paradigm, wrapping up network communication exposed by sockets, is based on extending the notion of *invocation* to a distributed context, i.e., by providing a form of remote invocation This paradigm presents many variations, which are regrouped *under* the term *remote procedure call* (RPC) [BN84, ATK92].

The RPC is probably one of the most popular paradigms for devising distributed applications.

2.2.1 From Remote Procedures to Remote Objects

Originally introduced for procedural programming models, e.g., *Sun RPC* [Sri95], *DCE RPC* [RKF93], remote invocations have been quickly applied to object-oriented languages, promoting some form of remotely accessible entities, such as *guardians* in Argus [Lis88] (follow-up of CLU [Lis93]), *network objects* in Modula-3 [CDJ$^+$89] and Obliq [Car95]. In the latter language, every object is potentially a network object. More recently, Java [GJSB00] introduces *remote objects* through its *remote method invocation* (RMI) [Sun99] paradigm. Also, interoperability has been added to remote invocations on objects through so-called *second-class* RPC packages (in contrast to *first-class* packages promoting a single language) such as the *Common Object Request Broker Architecture* (CORBA) [OMG01a], or the *Distributed Component Object Model* (DCOM) [Ses97]. This multiplicity is responsible for diverging terminologies, e.g., "request/reply", "client/server".

2.2.2 Proxies

The common abstraction underlying these approaches is best known by the name of *proxy* [Sha86]. In the original RPC implemented mainly in procedural languages, proxies where viewed as "empty" procedures representing effective procedures in remote address spaces. Straightforwardly, the proxy has been mapped in object-oriented settings to an object which mimics the remote object, i.e., it provides a similar (if not identical) interface to the one of the object it represents, and its *methods* (*member functions*, *operations*, etc.) are invoked locally as if the invocations were made on the original remote object. This is illustrated in Figure 2.3, by equipping the proxy with the interface of the original object. The proxy is mainly responsible for *marshaling* invocation arguments, that is, transforming them to a format more suitable for lower-level communication layers. The opposite action is called *unmarshaling*. Also here, many terminologies coexist. Proxies are sometimes also called *stubs* (e.g., [Sun99, OMG01a]), while their remote counterparts, responsible for unmarshaling, are usually called *skeletons*. Literature sometimes refers to skeletons as (server-side) stubs.

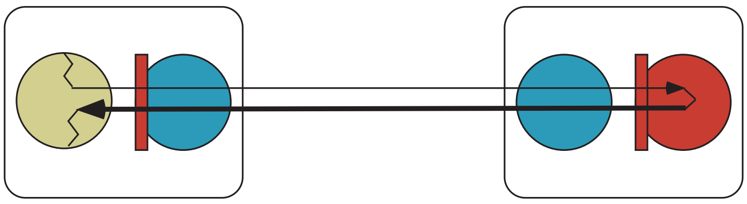


Figure 2.3: Interaction with Remote Procedure Calls and Derivatives

2.2.3 Synchronous RPC

By making remote interactions appear the same way as local ones, the RPC and its derivatives make distributed programming very easy. Remote objects are *synchronously* invoked through a proxy colocated with the invoker, and the thread of control resumes once the reply has been processed. The transparency offered by the RPC paradigm as a result of the exploiting of application-defined types explains the tremendous popularity of this paradigm in distributed computing.

Anthropomorphism: Holograms

Finding an illustration for synchronous interaction through a proxy requires more imagination, and we are forced here to have recourse to science fiction. "Holograms" have been extensively used in futuristic visions of communication between humans. A proxy can be compared to such a three-dimensional image which represents a far-away person. One could talk to the hologram, ask it whatever question, and expect a reply. Behind the scenes, one's own voice, and possibly also image, would be communicated to the effective partner (where this information might symmetrically be used to constitute a hologram). The hologram is nearly perfect, in the sense that it is very difficult to distinguish it from the real person.

Coupling

Remote interactions are however by their very nature different from local ones, e.g., by giving rise to further types of potential failures. This difference might be less visible in a *local area network* (LAN), but becomes more important once the involved participants multiply throughout a *wide area network* (WAN) such as the Internet, especially because the usual one-to-one semantics combined with a synchronous invocation cannot be efficiently used to disseminate information to several interested parties (see [BvR94] for a synopsis). Increased effort is required to efficiently implement invocations on several objects at a time (e.g., [BI93]).

As shown in Figure 2.3, the synchronous nature of RPC introduces a strong time, and space coupling since an invoking object requires a remote reference to each of its invokees, which has to be up at the moment at which the invocation occurs. In addition, the consumer[1] is strongly coupled in flow, due to the awaiting of the reply. Despite the inherent synchronization between consumers and producers (invokees), we do not consider the latter ones as coupled in flow, since they are usually not forced to explicitly "pull" for every invocation.

[1]The distinction of consumer/producer roles is not straightforward in RPC. We assume here that an RPC which yields a reply attributes a consumer role to the invoker, while the invokee acts as producer. As we will point out, the roles can be reversed with *asynchronous* invocations.

2.2.4 Asynchronous RPC

Several attempts have been made to reduce coupling in remote invocations, especially targeting at removing flow coupling in order to avoid blocking the caller thread on the reply of a remote invocation.

Oneway Call

A first variant of an *asynchronous* invocation consists in simply omitting return values in remote operations (e.g., CORBA *oneway*), as shown in Figure 2.4. This is often implemented as invocations with weak reliability guarantees (e.g., *best-effort*) because the sender does not implicitly receive success or failure notifications. This is opposed to synchronous RPC, where a failed invocation (from the invoker's point of view) will return incorrectly.

From unconnected sockets to oneway calls. Oneway calls can be viewed as a higher abstraction for asynchronous message sends (unconnected sockets), where the primitive used for sending displays an application-defined signature, and the arguments are automatically transformed to lower-level representations. Akin to unconnected sockets, the producer (here the invoker), and the consumer (invokee) are still coupled in time and space.

This ability of hiding underlying message passing mechanisms have motivated the integration of oneway calls in a series of object-oriented languages like ABCL/1 [YBS86] or Hybrid [Nie87]. Also, like further forms of asynchronous invocations discussed in the following, oneway calls suit very well the *actor* [Agh85] model of computation, e.g., Actalk [Bri89].

Anthropomorphism: a man's best friend. In this case, one can imagine a hologram which looks like the far-away person, but does not reply. Such a hologram can be used for instance to transmit indisputable orders to the receiver, or just to represent your best friend, whose behavior, while listening to your stories, can be fixed to simple gestures, in particular, nodding.

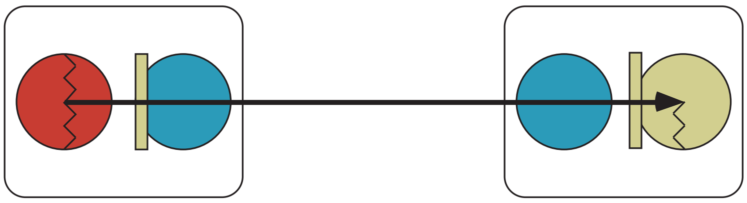


Figure 2.4: Oneway RPC

Explicit Futures

This second, less restrictive kind of asynchronous invocations, attempts to remove the inherent blocking of the consumer and hence flow coupling, yet by supporting return values.

Pull. However, instead of making an invoking thread wait for a result, a handle is directly returned upon the invocation. With this variant, commonly referred to as *future* (*future type message passing* [YBS86]), a client queries an *asynchronous object* for information by issuing a request to it. Instead of blocking however, the invoking thread can immediately proceed (Figure 2.5). Thanks to the handle, the client may query the result later (*lazy synchronization*, *wait-by-necessity* [Car93]), or ignore it. This mechanism has been successfully implemented in a wide variety of concurrent languages, e.g., ABCL/1, Concurrent Smalltalk [YT87], or also actor languages like Act++ [KML93].

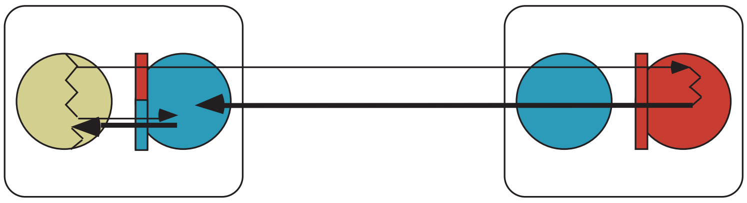


Figure 2.5: Asynchronous RPC Interaction with Pull-Style Futures

Callback. Some of these languages also implement callback mechanisms, to remove any form of flow coupling on the consumer side. Typically, when invoking an operation on a proxy, a callback object is provided as additional argument by the invoker, and is automatically invoked once the reply is ready (Figure 2.6). More recently, explicit futures with callback (as well as pulling flavors) have been integrated into the CORBA platform through the *Asynchronous Messaging Interface* (AMI) [OMG01a].

Anthropomorphisms: holograms for dummies. We can again use the hologram example to illustrate pulling futures. Such a hologram would not give a “real” reply to our queries, yet would for instance return a phone number that could be called later (or similarly a web address), to obtain the reply. Obviously, if the reply is not ready, we would only get the answering machine. This type of hologram could be ideally used to put off persons considered too nosy, or to avoid the honest but nevertheless infamous reply “I don’t know”.

To realize callbacks, we would give our mobile phone number to the hologram, such that we could be called back later, and the original reply of the remote interlocutor would be replayed to us.

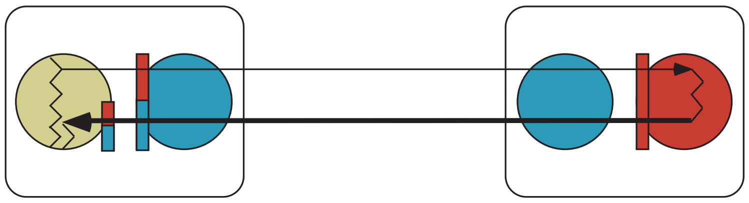


Figure 2.6: Asynchronous RPC Interaction with Callback Futures

Implicit Futures

This third style of futures tries to get the best of both the worlds of transparency and asynchrony. Remote objects are invoked in a seemingly synchronous style, but the thread of control can continue before the reply has been received or even computed, for as long as there is no attempt to effectively make use of the result in the subsequent code. Once the return value has been received on the client side, it is automatically "inserted" into the corresponding variable. Examples of languages including such implicit futures are Eiffel// [Car93], Karos [GCLR92], or Meld [KPHW89].

Avoiding (lazy) synchronization. This scheme, by all evidence, relies on a strong system support, and can be optimally exploited by using return values of remote invocations "as late as possible", since an attempt of using a return value before it has been received results again in blocking the corresponding thread. This constraint has to be kept in mind when devising applications based on implicit futures, but also with explicit futures supporting pull-style interaction. Indeed, many implementations of pull-style futures provide primitives for reading the future value which block the thread of execution if they are invoked "prematurely". Hence, also in this case, flow coupling is not systematically avoided. Implicit futures however still offer the advantage over such explicit futures that the signatures of methods provided by the proxies do not deviate from the original ones.

Anthropomorphism: holograms for smarts. This case requires even more imagination. Such a hologram would reply unclearly to us, making us believe that we have understood. At the moment when we would really need the information however, we would have to think about the previously received reply, and might even rely on information received in the meantime to complete the puzzle. This gives the questioned person time to communicate a complete reply, while giving the interrogator the time to understand, and the firm belief that the questioned had the answer ready from the beginning.

2.2.5 Notifications

With oneway invocations, a scheme similar to a callback future can be implemented by splitting a synchronous remote invocation into two oneway calls: the first one sent by the client to the server, accompanied by the invocation arguments including a callback reference to an object controlled by the client, and the second one sent by the server to the client to return the reply.

This scheme can be easily used to express several replies by having the server make several callback invocations on the client.

This type of interaction, where consumers register their interest directly with producers which manage subscriptions and send notifications, corresponds to a "distributed variant" of the so-called *observer design pattern* [GHJV95], illustrated in Figure 2.7. This communication style is often built on oneway invocations in order to enforce flow decoupling and corresponds thus more to a scheme or protocol than to a class of abstractions or paradigms. Though publishers asynchronously notify subscribers, both remain coupled in time and in space. Furthermore the communication management is left to the publisher and can become very burdensome as the system grows in size.

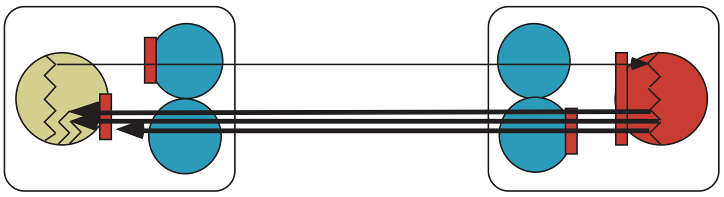


Figure 2.7: Notification-Based Interaction

2.2.6 From Proxies to Shared Objects

The original idea common to most implementations of objects globally accessible through proxies is that invocations made on those proxies are transmitted to the original object.

Smart Proxies

As widely noticed however (e.g., [WWWK94, Lea97b, Gue99]), remote invocations can be confronted with different behavior patterns than local ones, like increased latency, congestion, or link failures in the network. In reply to these constraints, and also to address further requirements (e.g., tracing, debugging), many RPC packages nowadays support configurable proxies (*smart proxies*) which can for each invocation, in addition to performing pre- and postinvocation actions, decide whether the operation should be effectively invoked on the represented remote object, or should rather be performed by another remote or local object. This also gives the proxy the possibility of playing the role of cache.

Distributed Object Implementations

The next step consists in distributing the effective implementation of the represented object, e.g., by replicating components. Such distributed shared objects (e.g., *Globe* [vSHT99]) aim at avoiding fully centralized components, and hence, single points of failure. Distributed shared objects are of application-defined types, and enable remote entities to indirectly interact. Providing a framework which automatizes the distribution of a custom object implementation is however non-trivial, and it is more convenient to use a more direct interaction between distributed address spaces based on specific predefined types of shared objects, such as the ones we will introduce in the next section.

Anthropomorphism: A Hologram for Hal

Smart proxies can be pictured as holograms generated by a computing device with some form of intelligence. Instead of sending a question to the remote person, the reply can be given by the computer itself,[2] or can be rerouted to another person.

A hologram representing a shared object could represent a somewhat “virtual” person, e.g., reflecting a computer-generated entire personality. Such holograms could be used for instance to ease the communication with computers manifesting human-like behavior, such as a substitute “best friend”, which besides nodding would also give simple replies like “yes”, “of course”, or even “you’re the best”.

2.3 Shared Spaces

An indirect interaction between distributed address spaces is achieved by the *distributed shared memory* (DSM) [PTM96] paradigm which can be seen as a specific type of shared object, representing a common shared memory. Like any kind of shared object, synchronization and communication between participants take place through operations on shared data.

2.3.1 Tuple Spaces

The *tuple space* underlying the *generative communication* style originally advocated by Linda [Gel85] provides a simple, yet powerful, distributed shared memory abstraction that is integrated with a language. A tuple space is composed of a collection of ordered *tuples*, equally accessible to all hosts of a distributed system.

Primitives

Hosts indirectly interact by inserting and removing tuples into/from the tuple space. Three main operations can be performed: `out()` to export a tuple in a tuple space,

[2]Such as “Hal” in Stanley Kubrick’s “2001: Space Odyssey”.

`in()` to import and remove a tuple from the tuple space and `read()` to read without withdrawing a tuple from the tuple space.

Tuples

A tuple inserted into the space is an arbitrary set of actual arguments. A tuple read from a space can be specified by actual arguments and formal arguments acting as placeholders. A candidate tuple must conform to such a tuple template in the sense that the types of the formal arguments of the latter one must match the types of the corresponding actual arguments of the former one, and the actual arguments of the latter one must match the corresponding arguments of the former one in type and value. In the original tuple space, tuple arguments were mainly values of *primitive types*, like integers, floats or character strings.

Concurrency and Coupling

This interaction model provides time and space decoupling in that tuple producers and consumers remain anonymous with respect to each other, and the tuple space acts as buffer between them. The creator of a tuple needs no knowledge about the future use of that tuple or its destination. An `in()`-based blocking pull interaction implements one-to-n semantics with *one-of-n* flavor, where only a single among several potential consumers reads a given tuple. In contrast, `read()`-based blocking pull interaction is able of providing one-to-n message delivery with some-of-n flavor, i.e., a given tuple can be read by all consumers whose criteria match that tuple. However, there is no flow decoupling since consumers pull new tuples from the space in a synchronous style (Figure 2.8).

Object Spaces

There have been a series of attempts to transform the structured form of tuples to an object form, mainly by extending the exact type equivalence for tuple elements in Linda to the notion of *type conformance* (see Section 3.4.1). While early approaches to integrating the tuple space with objects, like [MK88] (for Smalltalk [GR83]), promoted tuples as sets of objects, later approaches, like [Pol93] (C++ [Str97]), [Kie97] (Objective Linda), or [FHA99, CR97, LLW99] (Java) considered tuples as single objects, however often degrading their *attributes* (*fields*, *instance variables*, etc.) to tuple arguments (Section 3.1.2 gives more details on this).

Anthropomorphism: Bookstore

A tuple space manifests characteristics of both bookstores and public libraries. Information published in books or journals by authors, most probably personally unknown to the effective readers, can be bought (`in()`). In libraries, readings can be copied (`read`), or in the case of archives, can be printed. The bookstore only has a

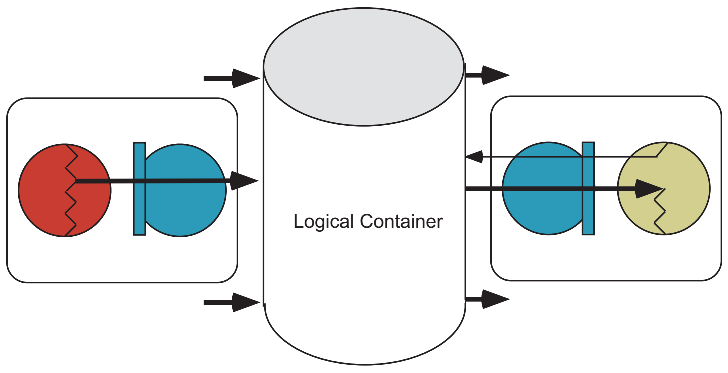


Figure 2.8: Interaction with Tuple Spaces

single copy of each book, and it is not a necessity that a new copy of a sold book will be reordered. In any case however, the reader personally picks up the desired piece of literature at the bookstore.

2.3.2 Message Queues

More recently, the shared space paradigm has been revisited by more industrial strength approaches, commonly termed *message queues*, e.g., [Sys00, BHL95, DEC94, Mic97, Ora99]. At the interaction scheme level, message queues recall much of tuple spaces, by representing global spaces which are fed with messages from producers.

First Come, First Serve

These queues promote a pull-style interaction of concurrent consumers with the queue, mainly in a way corresponding to the `in()` primitive found in tuple spaces (one-to-n with one-of-n flavor, also called *point-to-point* in this context). Which element is retrieved by a consumer is however not primarily defined by the element's structure, but by the order in which the elements are stored in (i.e., received by) the queue. Similarly to tuple spaces, producers and consumers are thus decoupled both in time and in space, but there is a lack of flow decoupling since consumers synchronously pull messages (Figure 2.9).

From a functional point of view, message queuing systems additionally provide transactional, timing, and ordering guarantees not necessarily considered by tuple spaces.

Anthropomorphism: Bookworms

Message queues can also be pictured as bookshops, with a rather specific type of clients, which are interested in any topic. Once the last book has been read to the end, another stop at the bookstore provides a client with a new reading.

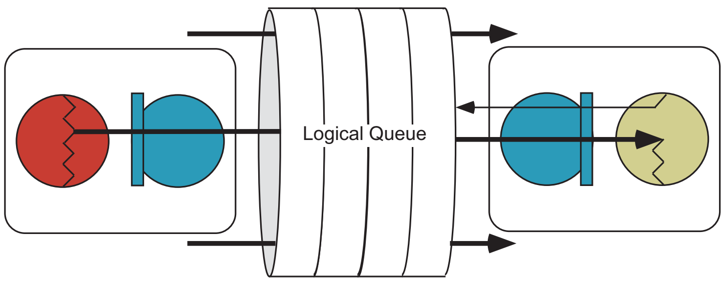


Figure 2.9: Interaction with Message Queues

2.3.3 Advanced Primitives

Tuple spaces have turned out to address the requirements of many distributed applications in terms of both interaction and synchronization. The latter possibility is mainly provided by the `in()`-style interaction, which integrates a powerful concurrency mechanism with communication, e.g., a form of *mutual exclusion* can be implemented with such an interaction style.

Improving Expressiveness

However, several primitives have been added to the original tuple space model, addressing certain flaws, like the lack for efficient *multiple reading* [RW96] (repeated invocations of the non-destructive `read()` primitive by a consumer can yield the same tuple, while other similarly eligible tuples might be ignored), and also several specific requirements. For instance, `eval()` is a more sophisticated primitive supporting functions as arguments of a tuple. These functions are evaluated, and are replaced by their return values in the tuple inserted into the space. Also, the use of *multiple tuple spaces* (see [Gel89] for the case of Linda) has been investigated to limit the scope for tuple spaces and their contained tuples.

Reducing Flow Coupling

More recently, several systems have introduced callback primitives aiming at removing the flow coupling on the consumer side, like *JavaSpaces* [FHA99] or also *TSpaces* [LLW99], both in the case of Java. Similarly, many message queuing systems integrate callback mechanisms. Such primitives are often provided with one-to-n semantics, leading to a publish/subscribe-like interaction scheme.

2.4 Publish/Subscribe

The publish/suscribe paradigm is characterized by the complete decoupling of producers (*publishers*) and consumers (*subscribers*) in time, in space, but also in flow.

As outlined above, the latter coupling is removed by calling back consumers. Table 2.1 summarizes the decoupling abilities of the different interaction styles.

2.4.1 From Shared Spaces to Publish/Subscribe

Publish/subscribe-based systems often provide an abstraction similar to the shared space describe above, i.e., giving the impression of a globally accessible data repository or software bus. Hence, publish/subscribe, together with message queuing often provided by database systems, are sometimes regrouped under the term *message-oriented middleware* (MOM).

Since we will however take a less industrial and *message-centric*, but rather a higher-level view of publish/subscribe as an interaction paradigm based on the notification of events between remote objects, we will not deal with the whole complexity of industrial strength solutions.[3]

A more precise discussion of the different debates related to the publish/subscribe paradigm and its incarnations and implementations can be found in [EFGK01], and an initial proposition for a unified terminology is the subject of [RW97]. A further taxonomy, emphasizing transactional features in publish/subscribe, can be found in [TR00].

2.4.2 Subject-Based Publish/Subscribe

Examples of publish/subscribe abstractions fostering the illusion of a ubiquitous software bus (Figure 2.10) are plentiful, e.g., *Information Bus* [OPSS93, AEM99, Col99, UM99], *Event Channel* [OMG01b, OMG00], *Connector* [Ske98]. Also, *Smart Sockets* [Cor99] allude to the similarity between multicast sockets and publish/subscribe.

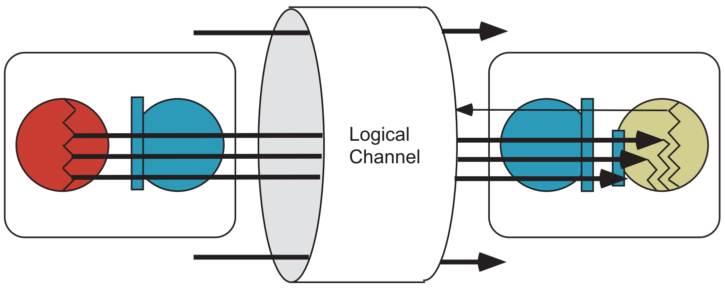


Figure 2.10: Interaction with Publish/Subscribe

[3]As illustration of the complex programming models of industrial systems, note that IBM has instrumented its ever famous *MQSeries* product with two API's, the *Common Messaging Interface* (CMI) and the *Application Messaging Interface* (AMI), as simpler alternatives to the original *Message Queuing Interface* (MQI).

Channels and Groups

To divide the event space, systems usually provide for several distinct spaces, or *channels*, by associating names with these. Such channels resemble much the *groups* [Pow96] known from *group communication* underlying for instance *replication protocols* [Bir93]: by subscribing to a group T one becomes member of the group T.

This scheme was implemented by nearly all early systems, and these groups also often appear under the name of *subjects*, leading to the widely adopted terminology of *subject-based publish/subscribe* implemented by many early systems. Examples include [Cor99, Ske98, TIB99, AEM99, EGS00].

Channels and subjects are often dissociated by the nature of their naming: in the context of channels, names are often viewed as network addresses like addresses of IP Multicast groups often built upon, while subjects are seen as represented by logical addresses (e.g., “MyEvents”). Even though IP Multicast sockets usually promote a pull-style interaction on the consumer side, they are sometimes pictured as implementing a publish/subscribe interaction [HGM01].

Subjects and Topics

Similarly, the term *topic-based publish/subscribe* is sometimes used as a synonym for subject-based publish/subscribe (e.g., [HGM01]), while some authors dissociate these two concepts by viewing the *hierarchical disposition* of names as reserved to subject-based publish/subscribe (e.g., [CRW00]). In any case, most approaches to publish/subscribe based on a group-like paradigm promote hierarchies with a URL-kind notation of group names, e.g., “/MyEvents/Today”. A subscription to a node in the hierarchy triggers subscriptions to the entire subtree, and wildcards can be used to perform pattern matching on subject names. All the above cited prototypes integrate such facilities.

Explicit Addressing

We believe the channel, topic, and subject abstractions to provide essentially equivalent *addressing schemes*, i.e., events are always published to specific groups. Any variant of group-based addressing strongly supports interoperability, by solely relying on names (character strings) to “connect” remote entities, especially in combination with events supporting some form of *introspection* (a form of dynamic queries, see Section 4.5.1) on their content. Such *self-describing events* (called self-describing messages in [OPSS93]), illustrated in Figure 2.11 with Java syntax, can be viewed as dynamic structures,[4] and are implemented by most current systems.

The dissemination is invariably based on a one-to-n model, mainly with all-of-n flavor, since when subscribing to a subject, one is interested in all events published for that subject.

[4]The *dynamic any* found in CORBA reflects the same design.

It is worth noting that sometimes space decoupling is considered as incompletely achieved by group-based publish/subscribe, since there is still an explicit notion of "address". Additions to the *subscription scheme*, like hierarchies, or *aliases* permitting the tagging of groups with different names [AEM99] make subscriptions more expressive, yet do not affect the basic addressing scheme. Any kind of interaction however requires a contract, whether this is an explicit group name, or a structural constraint put on the exchanged data.

Anthropomorphism: Paper Boy

Subscribing to a subject is like subscribing to a journal or newspaper. All subscribers receive a copy of every edition, which is delivered to them by the postman or a paper boy. The subject can be seen as the title of the journal, and, when viewing subjects as organized in hierarchies, one could also imagine that journals are classified according to keywords.

This illustrates the adequacy of publish/subscribe for mass-dissemination of information, as well as the difference to the shared space abstraction: when buying books from a bookstore, a consumer (unless already knowing the desired books and ordering them via Internet) has to pick these up in person.

```
public class DynamicallyStructuredEvent {
  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 {...}
```

Figure 2.11: Dynamically structured event

2.4.3 Content-Based Publish/Subscribe

Despite the improvements brought by subscription schemes based on hierarchies or aliases, any derivative of the group-based addressing style basically provides for an explicit division of the event space according to a single dimension. E.g., introducing two dimensions requires every possible value for the second dimension to be mapped to a nested subject in every subject representing a value of the first dimension. A more refined and expressive subscription scheme has been introduced through *content-based* (*property-based* [RW97]) publish/subscribe, e.g., [SAB+00, CRW00, ASS+99, CNF98].

Properties

Content-based publish/subscribe comes closer to its spiritual ancestor, the tuple space, by taking into consideration inherent *properties* of the conveyed data. Subscriptions are expressed as predicates based on these properties, and these *subscription patterns* are viewed as *filters* when (generated and) applied by the communication middleware. The explicit addressing known from subject-based systems hence disappears, and becomes implicitly given by the individual properties of events.

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. Dynamic structures, as depicted in Figure 2.11, can develop their full flavor with content-based publish/subscribe, by allowing attributes, and hence properties of events, to be initialized at runtime. Likewise, most standardized API's, like the *Java Message Service* (JMS) [HBS98], view properties as characteristics explicitly attached to events.

Expressing Subscriptions

Subscriptions can be viewed as issued to a channel, which similarly to a tuple space, can be a singleton, or when combining with subjects, can be a channel reflecting a specific subject. In any case, such a channel provides a one-to-n interaction style with some-of-n flavor. Subscription criteria consist of desired values for given properties, and can be expressed in various ways.

Query languages. Often, a subscription language is used to express *property-value pairs*. Examples can be found plentiful, e.g., the *Structured Query Language* (SQL), the OMG's *Object Query Language* (OQL) used in the *Cambridge Event Architecture* (CEA) [BMB+00], or the *Default Filter Constraint Language* used in the CORBA Notification Service [OMG00]. Relying on such pairs enables very efficient realizations, since computational overhead is reduced by allowing events to be represented and handled by indexed structures.

Arguments. Approaches like the *Component Object Model* (COM+) or also the CORBA *Event Service* and *Notification Service* make use of the proxy principle to implement a publish/subscribe scheme. An invocation made on a proxy of type T is performed on every subscriber of type T, and hence registering an object of type T means subscribing to any method defined by T.

To respect the asynchronous nature of publish/subscribe, proxies provide oneway operations. In contrast to the tuple space, where the `out()` primitive could take an arbitrary number of arguments of different types, the proxy principle allows the application to furthermore give own names to primitives. We will come back to this specific kind of publish/subscribe interaction in Section 3.7.2.

Templates. With tuple spaces, producers obtain tuples by providing a tuple to the `in()` or `read()` primitive whose arguments are not necessarily all initialized. Such tuples can be understood as *templates*, and most tuple space derivatives incorporating an asynchronous variant of the non-destructive `read()` primitive to promote publish/subscribe-like interaction are based on this approach.

Hence, when subscribing, a consumer provides a template object, which is matched with the potential objects of interest, for instance attribute-wise, or based on a specific `match()` method implemented by all events.

Anthropomorphism: Selective Readers

Content-based subscribing can be viewed as subscribing to individual journal editions. To be more precise, instead of subscribing to journals and only effectively reading some of the articles, (or even only some editions), one could imagine that a big publisher, like the ACM, or IEEE, would offer the reader the possibility of receiving only individual editions, yet of various journals (or even specifically composed ones), which contain for instance articles covering specific subjects, containing precise keywords, or which are written by our preferred authors or do not exceed a given length.

2.4.4 Event Correlation

Another face of publish/subscribe is called *event correlation* [KR95, MSS97, CRW00]. With event correlation, subscribers can express interest in being notified upon the occurrence of specific combinations of events only.

Subscription Scheme

Event correlation has been sometimes viewed as a publish/subscribe style of its own. We view event correlation as orthogonal to the main addressing scheme. For example, a subscriber could be interested in being notified of the occurrence of a pattern of events appearing under respective subjects. The subscription scheme however obviously depends on the addressing scheme, e.g., subscribing to hierarchically organized subjects requires events to be published to the same subjects.

In most systems based on event correlation, content-based subscribing is predominant. The individual events are merged, and the subscriber receives a compound event, which is composed of (subsets of) the attributes of each of the individual

Abstraction	Space Coupling	Time Coupling	Flow Coupling
Connected Sockets	Yes	Yes	Yes
Unconnected Sockets	Yes	Yes	Consumer
RPC	Yes	Yes	Consumer
Oneway RPC	Yes	Yes	No
Explicit Future (Pull)	Yes	Yes	No
Explicit Future (Callback)	Yes	Yes	No
Implicit Future	Yes	Yes	No
Notifications (Observer Design Pattern)	Yes	Yes	No
Tuple Spaces (Pull)	No	No	Consumer
Message Queues (Pull)	No	No	Consumer
Subject-Based Publish/Subscribe	No	No	No
Content-Based Publish/Subscribe	No	No	No

Table 2.1: Decoupling Abilities of Interaction Paradigms

events. Event correlation can hence be viewed as providing what we call a *m-to-n* (*many-to-many*) interaction style, since a single interaction can involve several producers and several consumers.[5] In the context of event correlation, subscription criteria are often called *event patterns* [RW97].

Anthropomorphism: Gourmet Readers

With event correlation, one can subscribe by providing all the criteria outlined in the case of selective reading, but will receive individually composed journals which only contain those articles which correspond to the specified criteria, these articles having been written and published at different moments.

[5]Note that often the "basic" publish/subscribe paradigm, without event correlation, is said to provide many-to-many interaction. According to the taxonomy adopted in this thesis, which focuses on single interactions, i.e., single data exchanges between consumers and producers, event correlation is the only interaction paradigm to involve several producers and consumers into the same interaction. In fact, event correlation provides m-to-n semantics with *some-of-m* and some-of-n semantics, when a content-based subscription style is chosen.

Summary

Distributed computing would be hardly conceivable without any form of sockets. This fact is conveyed by the inherent support that this paradigm benefits from in prevalent operating systems. Akin, RPC mechanisms have been integrated into most operating systems. Forms of shared spaces are viewed as foundations for parallel, concurrent, and distributed programming. In other terms, all these paradigms have proven their immense practical value in respective contexts, for which they have been designed.

The publish/subscribe paradigm has been motivated by scalability requirements of todays applications in wide-area networks like the Internet. The excellent scalability properties offered by publish/subscribe at the abstraction level can be explained by its strongly asynchronous flavor, ensuring a strong decoupling of participants in *time* (participants do not have to be up at the same time), in *space* (participants do not have to know each other), and in *flow* (the main flow of control of participants is not blocked).

Chapter 3

Type-Based Publish/Subscribe (TPS): Concepts

Common approaches to publish/subscribe interaction altogether fail in sufficiently providing *type safety* with *application-defined event types*, and expressive content-based matching based on these types, while preserving *encapsulation*.

The type-based publish/subscribe (TPS) paradigm addresses these requirements. Events are viewed as objects, i.e., instances of application-defined object types. Subscribers outline their preferences based on the types of the desired event objects (*obvents*), along with more refined content-based queries based on the public members of these event types.

TPS does not require any explicit notion of event kind, nor any explicit addressing scheme, but relies solely on an “ordinary” type system common to all event types, e.g., that of an object-oriented programming language, to describe subscriptions. By basing subscriptions on the types of the events, the type of the received events is known, and type checks can be performed.

This chapter presents the TPS paradigm for distributed programming in a general manner. Increased attention is however given to the Java [GJSB00] programming language, in whose context we will describe TPS in the two following chapters.

3.1 Object-Oriented Publish/Subscribe

While the implementation of message passing, RPC, and shared spaces in an object-oriented setting have been thoroughly studied, only comparably little effort has been made to merge the benefits of event-based distributed programming based on publish/subscribe with those of the object paradigm. Current publish/subscribe models are inadequate for object settings, and we outline their flaws, mainly with respect to type safety and encapsulation.

3.1.1 Type Safety

Very few approaches to publish/subscribe permit the use of application-defined event types. Events are often viewed as low-level messages, and a predefined set of such message types are offered by most systems, which provides for very little flexibility.

Dynamic Structures

To repair this lack of flexibility to some extent, many systems propose a variant of self-describing events. These indeed increase flexibility, and are very useful in certain scenarios where dynamic interaction is required. Nevertheless, just like invocations interpreted dynamically through the *dynamic skeleton interface* in CORBA (or similarly in DCOM) are far less often encountered than plain static invocations, the systematic use of self-describing events can become cumbersome, and in addition jeopardizes type safety.

Only the Best is Good Enough

Promoting events as instances of a low-level data type like a byte array can easily lead to an API which appears type-safe. From the application point of view however, such an API is weakly typed, since application objects have to be explicitly transformed to and from such low-level types. We do not consider such an API as type-safe, and this remark can be extended to self-describing events and any kind of event or message promoting a non-object abstraction.

Also, any event abstraction which necessarily entails explicit type checks and casts when using own event types is here considered as not type-safe. We hence focus mainly on static typing, in the sense that we assume that, in a language which enforces static type-checking, these same checks should be applied to any remote communication, in our case, to the exchange of events.

Application-Defined Types

As already successfully shown by RPC packages, type-safe distributed interaction can be indeed achieved with application-defined types. Such application-defined types can nonetheless express their distributed nature, for instance by inheriting "distributed behavior" from predefined basic types.

3.1.2 Encapsulation

Existing approaches which enforce application-defined event types offer API's which do not take these types into consideration, with a negative effect on type safety. For example, such systems provide an API where formal arguments representing events appear as instances of a basic event type T, while the use of instances of any subtype

T' of T implies casts from type T to that more narrow type T', if features added by type T' have to be accessed.

Furthermore, these types are most commonly viewed as sets of attributes, though promoted as objects, and this unconditionally leads to violating encapsulation in subscriptions and filtering. Note here that we do not abolish any direct access of attributes, but try to avoid attributes as the *only* possibility of describing event properties (see Section 3.5.2).

Query Languages

Often, subscription patterns are based on *attribute-value pairs*, and are expressed through SQL-like query languages. Besides violating encapsulation, type safety is jeopardized again, since such queries are usually expressed through strings which must be parsed at runtime. Increased portability is a flawed argument for query languages, given the multitude of different languages.

Templates

Implementing template-based subscriptions with attribute-wise matching again violates object encapsulation, since the developer is constrained to having this scheme in mind, and devising objects as sets of publicly accessible attributes. Moreover, template-based subscriptions offer only limited expressiveness: attributes are verified for strict equality with a value. Hence, it is not possible to match an attribute against a range of values. Also is it impossible to express a constraint not on an entire attribute, but recursively only on one of it's attributes. Implementing the matching inside the template object rules out any performing of optimizations by the middleware, such as regrouping filters and factoring out redundancies.

Optimizations

In fact, as illustrated by *Gryphon* [ASS$^+$99], the stifled scalability of content-based publish/subscribe systems resulting from high expressiveness can be counteracted by exploiting common interests of subscribers to avoid redundant filtering and routing. In subsequent discussions, the importance and impact of such optimizations will be devoted more attention.

3.1.3 Requirements Overview

By attempting to merge the benefits of objects and event-based distributed programming based on the publish/subscribe paradigm, this thesis precisely makes a case against the argument that type safety and encapsulation are inherently incompatible with the publish/subscribe paradigm. More precisely, TPS addresses the following requirements:

Type safety: Distributed applications, by their nature, tend to become extremely complex. Type safety helps counteracting potential errors resulting from this inherent complexity, and should be enforced by any distributed programming abstraction.

Encapsulation: Events are to be viewed as objects, and the encapsulation of these objects should not be systematically violated through fine-grained (content-based) subscriptions. In particular, the expression of subscription patterns should not impose viewing events as sets of attributes.

Application-defined event types: Applications should be free to define their own event types, and should not have to abide to a restricted set of predefined types. Such predefined event types can indeed assist developers, but should not be the only admissible event types.

Qualities of Service: Different applications have different requirements in terms of guarantees ensured by the communication medium. The latter component should provide different semantical choices, reflecting *Qualities of Service* (QoS) typical for distributed contexts.

Optimizable subscriptions: Subscriptions should not be opaque to the communication middleware, allowing it to apply optimizations when performing the actual routing and filtering of events.

Several similar notions of typed publish/subscribe exist in the literature, yet none succeeds in respecting all these requirements. We overview the closest related approaches in Section 3.7.

3.2 Obvents: Marrying Events and Objects

To depict more precisely how type-based publish/subscribe (TPS) unites the two worlds of event-based distributed interaction and object-orientation, we introduce an object model which distinguishes two main categories of objects. The distinction is coarse, and is made very roughly according to the granularity of these objects. The resulting model strongly resembles the model presented by Oki et al. in the context of subject-based publish/subscribe [OPSS93]. The added value of the object model presented here in the context of TPS consists in the introduction of two further (sub)categories of objects.

3.2.1 Unbound Objects

Unbound objects are *locality-unbound*, that is, their semantics do not depend on any local resource, and they are easily relocatable (*data objects* in [OPSS93]). In other terms, these objects could at any moment be easily transformed into a representation more suitable for lower-level communication layers, e.g., a byte array, and transferred to another address space.

Fine-Grained Objects

In practice, these objects are "small". The most simple, yet realistic, examples are simple object types encapsulating a value of a primitive type, whether these are *primitive object types* provided by the language (e.g., Java, Smalltalk, Eiffel [Mey92]), or explicitly defined. Nevertheless, most unbound objects offer a complexity which usually exceeds that of a single attribute of primitive type. It is very common to have unbound objects encapsulate several attributes of primitive types, or even objects which again encompass attributes. Again, since such objects have to be transmitted over the wire, their size should be "reasonable": although appealing, the goal is not to provide a general service for object *migration*.

Such objects can be conveyed with various communication schemes, according to the desired interaction style: they can be objects representing tuples, as used with tuple spaces (Section 2.3)), or invocation arguments for RPCs which are passed by value.

Obvents

Another specific kind of unbound objects are events. Such objects are used to notify events, and we consider them as first-class citizens defined by the application.

Events can be represented as first-class citizens in mainly two ways, either (1) by introducing specific object types reflecting events, or (2) by promoting specific constructs. Our main goal is to make events come as close as possible to "arbitrary" objects, and we will hence not promote events as specific constructs. This is as adequate to achieve type-safe event-based distributed interaction as adding specific constructs, while the latter approach unconditionally entails extensions to any candidate language. To emphasize the fact that events are objects, these will be henceforth called *event objects*, or to abbreviate notation, simply *obvents*.

3.2.2 Bound Objects

The second rough category of objects are *locality-bound*, i.e., they are rather coarse-grained objects which are bound to an address space and remain in that address space during their entire lifetime. They are termed *service objects* in [OPSS93].

Though such objects are very likely to make use of local resources, like any kind of human-machine interface, or simply the file system, these objects do not necessarily rely on a unique host, that is, such objects are not necessarily tied to a *particular* address space for geographical reasons.

Coarse-Grained Objects

The distinction between bound and unbound objects goes more along our criteria of relocation, based on the granularity of objects: bound objects cannot be easily migrated, or in other terms, bound objects are not bound because they *could not be* *running* in another address space, but because they*could not be transferred* easily to another address space.

Such objects are typical candidates for becoming objects which are remotely invocable through RPC-style interaction. To support RPC, such objects are made remotely "visible" by having them export a remote interface, through which they can be directly invoked from another bound object. More generally, bound objects communicate with other bound objects through various forms of remote communication, mainly by exchanging unbound objects. Typically, *components* in the sense of *component-oriented programming* (e.g., Java Beans [Tho98], COM [Obe00]) can be viewed as bound objects. Note in this context that event-based communication based on the publish/subscribe paradigm has been largely adopted by such component technologies, because of its strong decoupling of participants [EGS01].

Subscribers

In particular, when interacting in a publish/subscribe style, a bound object can take the role of subscriber.

Similarly to event objects which can appear as "normal" objects, or as specific constructs with limited semantics from the object point of view (maybe additional semantics as event constructs), it is not a necessity that subscribers be reflected by objects either. As we will depict in Chapter 5, subscribers can be viewed as pure code, e.g., a procedure.[1]

Subscribers are not only bound because of their size, but also because they represent a "role", a component's participation in a publish/subscribe-style interaction. A subscriber is always an incarnation of such a role, and a subscription could be migrated without migrating the corresponding subscriber object, since such an object is merely viewed as an incarnation.

3.3 Publishing Obvents

To avoid reduce synchronization between participants, we have adopted straightforward semantics for publishing obvents. Alternatives and dangers are discussed in Section 3.7.3.

3.3.1 Semantics

In short, when an obvent is published, every subscriber receives a copy of that obvent,[2] the state of these copies being the state of the original obvent at the moment of the publication.

[1]Any function can be represented by an object on which the function is defined. Inversely, an object can be captured by a function which returns that object.

[2]For simplicity we suppose reliable communication here.

More precisely, a distinct copy of a published obvent is created for each subscriber, according to the following rules:

Obvent global uniqueness: Consider an obvent o published from an address space a_1: if an address space a_2 contains two subscribers s_1 and s_2, these will receive references to two new distinct clones of o, say o_1 and o_2. The obvent o can be published several times, giving rise to several distinct clones in both a_1 and a_2.

Obvent local uniqueness: In the above scenario, if the address space a_1 also contains a subscriber s_3, then s_3 will receive a reference to a new obvent every time o is published.

Alternatively, one could also suggest that a *single* copy of a published obvent is created for each address space hosting subscribers. However, every subscriber (whether colocated with other subscribers or not) which receives a given obvent might have different tasks to perform upon reception of that obvent. In particular, these tasks might also involve the modification of the obvent, which would require additional effort to ensure synchronization between subscribers.

3.3.2 Publishing in Perspective

The action of publishing an obvent manifests flavors of two primitive actions commonly defined on objects, namely *object creation*, and *object cloning.*

Distributed Object Creation

The action of publishing an event object o can best be pictured as a distributed form of object creation (`new` keyword in a vast range of prevalent object-oriented programming languages), where o acts as template. The set of address spaces where this action will take place is given by the set of address spaces who are willing to host such objects, i.e., who contain subscribers whose subscription criteria match the template object.

An obvent class can hence best be pictured as a factory for instances incarnating notifications for events of the same kind, i.e., from the same *event source.*

Distributed Object Cloning

Similarly, with a published obvent o acting as template, such a publication can be pictured as a distributed *object cloning*, where a clone of the object o is created for every subscribed object. A subscription thus can be viewed as expressing the desire of getting hold of a clone of every published object which corresponds to the subscription criteria.

Note that in this context, the act of cloning corresponds to the term *deep cloning* used in [GS00]: when a clone of an object is created, its attributes are recursively cloned, i.e., a distinct copy is created, and not only a new reference to the respective object.

3.3.3 Sending Obvents over the Wire

The deep cloning described above is a consequence of the remote nature of a publication: to create a copy of an obvent o in a distinct address space, o, or rather a representation of its state, must be transferred to that address space, where a new instance can be created and initialized with the same state. Therefore, clones of an obvent cannot contain references to objects pointed to by the published obvent. Note that if an attribute of a cloned object represents a proxy object (e.g., Java RMI), a clone of that proxy can be created, however still giving access to the original remote object.

Object Serialization

The action of recursively traversing the attributes of an object and writing these to a more low-level representation is commonly called *serialization*, its antagonist is called *deserialization*. The targeted data type can be a byte array, or a stream.

In practice, such serialization, resp. deserialization, of obvents can be rather easily provided for objects representing primitive types, but requires more effort in the case of nested application-defined types.

Note here that the terms *marshaling* and *unmarshaling* widely employed in the context of RPC (cf. Section 2.2), are rather used to denote the serialization and deserialization applied to entire sets of objects, e.g., representing a list of arguments for a remote invocation.

Built-In Serialization Mechanisms

Certain languages offer built-in mechanisms to ease the serialization and deserialization of application-defined types. For instance, Java has been inspired by the Smalltalk model, and provides a basic serializable type which an application type can subtype, making its instances serializable and deserializable without much effort. Smalltalk, just like Java, offers a singly-rooted type hierarchy, yet integrates the basic serializable type with that very root, making every object serializable by default. In contrast, Java introduces a specific type `java.io.Serializable` as parent for all serializable types. Java however often "fakes" a serializable root by offering interfaces to core libraries where serializable objects are expected, but formal arguments are of the very root object type (`java.lang.Object`), leading to potential exceptions at runtime.

Instrumenting Objects with Serialization

In languages lacking such types, explicit types can be introduced as parents for all serializable types, with specific virtual methods which have to be implemented by the developer, e.g., an instance method `serialize()` to obtain a low-level representation from an object, and a class method `deserialize()` to create an instance from a low-level representation. In languages lacking class methods, e.g., Eiffel, an "empty" instance can be first created with an argument-less constructor, and initialized through an instance method `deserialize()`. To decouple classes from their serialization and deserialization, a design pattern such as the *factory method pattern* [GHJV95] can be chosen. If transformation into different data types is required, the *serialization pattern* [RSB^{+}98] is an interesting alternative.

3.4 Subscribing to Obvent Types

By using the types of obvents as *basic* subscription criteria, TPS strongly enforces the melding of a publish/subscribe middleware with the application: by matching the notion of *event kind* with that of an *event type*, i.e., using a "conventional" type scheme as subscription scheme, the type of the received events is known, and type checks can be performed. We discuss TPS in the face of different type systems.

3.4.1 Background: Conformance and Subtyping

By subscribing to a "type", a subscriber manifests its interest in *all* instances of that type. This includes *any* event object which *conforms* to that type, that is, also instances of any distinct type which is through some relationship defined as part of the considered type system, compatible with that type.

Conformance

Basically, a type system consists of mechanisms for describing sets of values (types) and a set of conformance rules (type-checking rules) for testing compatibility between types. The conformance rules define a substitution rule for values. Basically, if a type T_2 conforms to a type T_1, then any instance of type T_2 can be used in a consistent manner anywhere an instance of type T_1 is expected.

Subtyping

Usually, the set of types which conform to a type T_1 somehow include the characteristics of T_1. We will henceforth make use of the widely adopted term *subtyping* to describe the relationship between descending types, and denote it with the symbol $\leq$. A type T_2 which conforms to a type T_1 is a subtype of T_1, written $T_2 \leq T_1$, and $\forall\, T$, $T \leq T$. According to the two main types of conformance, namely *nominal* (*explicit*, sometimes also *inheritance*) conformance and *structural* (*implicit*) conformance, two kinds of subtyping can be defined.

Nominal subtyping: With nominal subtyping, a type T_2 inherits from a type T_1, meaning that T_2 *explicitly* inherits T_1's characteristics. T_2 is thus a subtype of T_1, and this is usually declared in the inheriting type (here T_2), but can also be made in the inherited type (T_1), allowing to add new *supertypes* to existing types like in Cecil [Cha95] or Sather [SOM94].

Structural subtyping: With structural subtyping, a type T_2 which offers the same public members (and more) than a type T_1 is said to be a subtype of T_1. The conformance relationships are hence viewed as *implicitly* defined by structural equivalences based on collections of signatures: any type which supports the required signatures can instantiate the parameterized declaration.

Subtyping in TPS

In general, the term "subtyping" is often used to express the possibility of "updating" existing types, that is, either extending them or defining compatible alternatives. Following that interpretation, subtyping is a well-known paradigm which offers a powerful means to address the requirements for extensibility, reusability, and compatibility of software.

With TPS, one can benefit from a straightforward application of these concepts to a *distributed* context, e.g., TPS gracefully enforces the extensibility of *distributed* applications: akin to centralized contexts, distributed applications can be updated by adding new subtypes of obvent types, in such a way that components which do not have to react to the extensions remain fully operational.

Adding new Subtypes

Obviously, such an updating at runtime can only be performed with the a priori permission of the language environment. *Dynamic class loading* has received an increasing attention through the rapid proliferation of Java, which inspired by Smalltalk, is based on the *virtual machine* concept. Conceptually, the Java virtual machine has, besides proven to provide an ideal compromise between source code interpretation and compilation, and thereby also portability and performance, addressed the increasing demand for continuous updating in a rather unique way [LB98]. While for instance Oberon [Rei91] or CommonLisp [Jr.90] have similarly introduced dynamic loading and linking, they do not provide for type safety, user-defined class loading policies, and multiple namespaces with lazy loading simultaneously. Dynamic languages such as Lisp [MAE^{+}65], Smalltalk and Self [US87] achieve type safety through costly runtime type checks.

Strongly inspired by Java, C# [Lib01], is one of the most recent programming environments to promote dynamic class loading as integral part of its programming model.

3.4.2 Obvent Type Systems

In any case, and quite obviously, the semantics of type-based subscribing strongly depends on the considered event type scheme. Such a type scheme can be given by the programming language, leading to a first-class middleware in the sense of first-class RPC packages which only consider a single language (e.g., Java RMI [Sun99]). In practice, in this case, it is sometimes easier to consider only a subset of the language's type system, since such type systems are usually designed for a "local" use, and a distributed deployment might introduce conflicts. For example, *scopes* of types have a strong impact, as we will discuss shortly.

Alternatively, such an obvent type scheme can be specific for obvents, in order to address requirements that differ from those for unbound objects (e.g., structural conformance), and in order to aim at a second class TPS package, i.e., a language-interopable approach. If the resulting type system is not at least a subset of every considered language's types system, this might require language modifications, e.g, by introducing events as specific first-class constructs, without connection to the language's very type system (e.g., ECO, see Section 3.7.2).

The goal here is not to introduce the perfect type system, since there are several tradeoffs between desirable properties (see [MMMP90]), leading to domain-related solutions prioritizing certain of these properties. We rather aim at illustrating several possibilities, pointing out the added complexity of devising a "distributed" (event) type system. In particular, we focus on possible pitfalls when applying a type system conceived without distribution in mind to a distributed setting.

A "Classic" Type System: C++

In certain languages based on nominal conformance, like C++, Eiffel, or Modula-3 [CDJ$^+$89], the code inheritance relation at the same time determines the type conformance (subtype) relation. In such type schemes, the notions of *type* (*abstract type*, *type definition*, *interface*, *signature*) and *class* (*concrete type*, *type implementation)* are identical, and accordingly subscribing to a type means subscribing to all instances of the indistinguishable class and its inheriting subclasses.

C++ Classes. C++ applies the merging of types and classes straightforwardly. The features offered by C++ include roughly the following:

Multiple subtyping: Multiple subtyping is provided through multiple inheritance: a class can subtype one or several classes, making of its inherent type a subtype of the types corresponding to the superclasses.

Abstract classes: Methods can be defined as abstract, that is, without method body. Classes which contain at least one abstract method are called *abstract classes.* These cannot be instantiated. Such an abstract class reflects an abstract type, but can nevertheless be a subclass of a non-abstract class.

Parametric polymorphism: Classes can be parameterized by types (see Section 4.2.1 for more details). In C++, such classes are called *template classes*.

Friends: Classes can be defined as *friends* for single methods or entire classes, giving access to otherwise hidden methods.

Nested Classes: C++ also offers *nested classes*,[3] which are classes that are contained in enclosing classes. The containment relationship of such nested classes affects from our perspective mainly their naming, and poses no particular problems in our context, since a nested class and its containing class do not have any particular links to each other. This is different in Java, as we will show later.

Aggregate Types. C++ has inherited also a set of *aggregate types* from its ancestor C. In short, structures (*structs*) are a shorthand notation for classes whose members are public by default (in contrast C structures represent data structures without associated functions), *arrays* can be defined for any type, and *unions* enable the expression of variants, i.e., a union can contain a value of several different types. Albeit, at any moment, a single member is populated in such a union, and its size in memory is thus the size of its largest member.

C++ also provides primitive types, which are however not relevant in our case, since we are mostly interested in application-defined (event) types.

Inheritance. Though presented here as a "classic" type system, the C++ type system presents potential conflicts when applied to obvents. We illustrate this here through the different types of inheritance found in C++. Classes can inherit their superclasses in a *public*, *protected*, or *private* way. Private inheritance implies that the potentially visible (public and protected) members can only be accessed by the inheriting subclass and its friends. A protected inheritance means that furthermore the subclass's recursive subclass and the latter one's friends can access the visible members of the inherited class. In particular, instances of a class C_2 which inherits in a protected or private way from a superclass C_1 can not be cast to its supertype outside of the class C_2 itself or its friend classes. As a consequence, an instance of such a class C_2 cannot be assigned to a variable of the supertype C_1 at any moment (see p. 743 in [Str97]).

"Distributed" Scope. In the above scenario, a type coercion can however very well take place inside the inheriting class C_2, and such a cast instance can be passed to the outside world. When such a cast object is then published, whether inside the class C_2 itself or after being passed outside as depicted above, the handling becomes more delicate. Indeed, the object will be serialized, and later on deserialized. The deserialization mechanism must be devised in a way which allows any class to deserialize instances of C_2, which in this case leads to creating an instance of C_2, yet using it as an instance of C_1.

[3]Following [Mad99], we will avoid the term *inner class*.

The object which performs the deserialization of instances of C_2 is not automatically entitled to cast them to the supertype C_1, making the deserialization impossible if the deserialization mechanism has not been conceived as depicted above. The nature of this problem lies in the now "distributed" scope of types. One way to solve this particular problem, could consist in forcing any obvent class to declare a given class C, which would then perform the cast, as friend. C would then be the class responsible for deserialization of the corresponding obvent class.

Alternatively, a more general approach could consist in using a subset of the semantics of the type system for obvent definition, e.g., restricting the use to globally public types for obvents. In this case, like in many others, most pitfalls can be avoided. A precise consideration of all singularities of the type system of a language is however in any case advisable.

Separating Type Definition and Implementation: Java

In an often cited paper, Cook et al. [CHC90] advocate for a clear separation of inheritance and subtyping. Roughly, inheritance is viewed as a mechanism for behavior sharing between classes (code reuse), while subtyping is pictured as defining conformance rules between types, which can be independently implemented by classes.

Examples of Type Systems with Separation. Several languages have advocated such a separation, like Objective-C [CN91], where *protocols* define abstract types, and classes implement these, or Sather, where multiple subtyping and subclassing are similarly featured in different hierarchies. Rather recently, Java has made this concept of separating types and classes more popular, through a more simple, nonetheless flexible type system.[4]

Primitive Types. Inspired by C++, Java offers primitive types, like `float`, `int`, `boolean`. The Java runtime environment however additionally defines corresponding primitive object types, e.g., `Float`, `Integer`, `Boolean`, in the core package `java.lang`.

Application-Defined Types. Java offers only simple inheritance, avoiding any problems known from multiple inheritance, yet introduces multiple subtyping through interfaces. In Java, types can be defined in the following two ways:

Explicit definition: A type can be explicitly declared by declaring an interface, which can subtype several superinterfaces: an interface I_1 which extends another interface I_2 represents a subtype of the type declared by I_2.

Implicit definition: Defining a class C implicitly declares a type, and at the same time gives the class which implements it. If a class C_2 inherits from another class

[4]Note that several solutions have also appeared to augment existing programming languages that lack this separation, e.g, by introducing signatures for C++ [BR97].

C_1, then the type defined by C_2 is a subtype of the type of C_1. 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. Just like in C++, a class can also be made abstract, by omitting at least one method body and tagging the corresponding method(s) as abstract.

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

As a consequence of this duality of types, it must be possible in Java to subscribe to interfaces as well as to classes.

Nested Java Types. Java supports nested types, which means that a type can be declared within another. There are four different ways of declaring nested types in Java.

Anonymous classes: These are classes that have no name. They combine the syntax for class definition with the syntax for object instantiation. Such classes cannot be referenced and can thus not be directly subscribed to.

Local classes: A local class is defined within a block of Java code, and is visible only within that block. It can therefore not be defined as public. The only classes that can inherit from such local classes, are other local classes defined in the same body.

Static member classes and interfaces: A class (or interface) of this kind behaves much like a top-level class (or interface), except that it is declared within another class or interface. A static member class can access *static fields* (*class variables*) and methods of the containing class which would otherwise be hidden.

Member classes: This last kind of nested class is much like the previous one, except that an instance is associated with an instance of the class in which the member class is declared. All fields and methods of that associated instance are thus accessible as well. Note that for such classes to be serializable, the containing class must be serializable as well, since the associated instance of the containing class must be transferred with the member class instance in order to always be accessible to the latter one.

Subscription Through Subtyping. A similar question to the one posed in the case of C++ arises here: what happens if a type T_2 is declared as subtype of a public obvent type T_1, but with a restricted scope? T_2 can here be a nested type with restricted scope, but also a type with package visibility. If instances of T_2 are published, they should be received by any subscriber which advertised interest in the public supertype T_1.

While C++ does not define any serialization mechanism, requiring care when devising an "own" mechanism which takes into account such special cases, Java defines its own serialization mechanism, which enables the deserialization of an instance of a class which would not be visible at that moment.

Subscriptions to Nested Types. Another issue is the handling of direct subscriptions to nested types. The case of anonymous classes in Java is fairly simple. Since these are not named, they cannot be subscribed to.

In general, one must also consider that subscriptions to any type with a restricted scope obviously only make sense if it is possible to publish any instances of that type (or of any subtype). At a first glance, instances of local classes in Java can only be published from inside the code fragment in which they are defined, which at the same time limits their subscriptions: subtypes of these local classes can only be declared in the same code fragment, which is also the only place where direct subscriptions to such types can be issued. This makes the usefulness of supporting such local classes in TPS questionable. At a second glance, many awkward situations appear with the *reflective* capabilities of Java (see Section 4.5.1 for details). In the case of local classes, a "reference" to that class can always be passed outside of the method body declaring that class. That way, it is possible to create instances from outside of the method body defining that class.

The same can be done in the case of (static) member types. However, a public (static) member class C_2 can always inherit from a private member class C_1, and hence instances of C_1 can always be published from outside of the declaring class, even without making use of reflective mechanisms. Supporting such types in TPS thus makes more sense.

Interoperable Events: Event Definition Language

Supporting a distributed communication paradigm in a programming language is often misinterpreted as limiting the use of a middleware platform to a single language. As however successfully shown by CORBA, interoperability can indeed be provided by integrating the middleware with *several* programming language*s*. The case of RPC, the main interaction paradigm in CORBA, is not less tedious than publish/subscribe, since it relies as much on the type systems and furthermore invocation semantics of the various languages than TPS does.

Invocations. In CORBA the problem is mainly solved by introducing a neutral *Interface Definition Language* (IDL) for types of remotely invocable objects. This leads to a language-independent type system with mappings to concrete programming languages, such as Java, C++, Smalltalk, Ada [Ada95], but also procedural languages like C and Cobol; even the functional language Lisp. Along its predefined primitive types, the IDL serves as the glue between different languages and platforms. In particular, it identifies how invocations pass between different languages.

Objects. In the case of TPS, not only object references disguised as proxies are passed from one address space to another, but entire obvents. Also concerning this issue, the OMG has cleared the path: support for *value types* is also provided by CORBA since version 2.3, allowing "arbitrary" objects to be passed by value between different languages. There is however no magic behind the adopted solution. Such value types have to be implemented in all potential languages and environments, and factories for serialization and deserialization are required.

Also, the development of Microsoft's *Common Language Runtime* (CLR) [TT01], an inter-language virtual machine, has brought out the *Common Language Specification* (CLS), which gives hints on the feasible semantics of an EDL, and also how to enforce interoperability between objects of different languages.

Obvents. The part of the CORBA IDL specification concerning such value types (probably even a subset) could be reused as an *Event Definition Language* (EDL) to help adding interoperability to TPS. Proposals for similar languages are plentiful, like the ODMG's *Object Definition Language* (ODL) used in the Cambridge Event Architecture [BMB$^+$00], languages used in tuple space implementations, e.g., *Object Interchange Language* (OIL) in Objective Linda [Kie97], or more recently, also the *eXtensible Markup Language* (XML).

100% Pure Content: Structural Conformance

As outlined previously, certain programming languages promote structural conformance, and hence structural subtyping, meaning that a type T_2 is implicitly a subtype of another type T_1, if T_2 provides at least the public members of T_1, with conforming signatures.

Examples of Type Systems with Structural Conformance. Examples of object-oriented languages with structural conformance include Amber [Car86], early versions of Strongtalk [BG93], Cecil and. Theta [LCD$^+$95] (based on so-called *where clauses* originally proposed in CLU). Several extensions for languages like Java have been proposed, e.g., [MBL97, LBR96], and certain authors also describe interesting mixtures of structural and nominal subtyping, like the PolyTOIL language [BSvG95], or an alternative type system for Java [BW98].

In Distribution. The advantages of structural conformance appear clearly, especially in a distributed context where different components have to be "glued" together: one cannot expect vendors to agree on basic types, and the resulting interfaces are by nominal conformance definitely incompatible.

RPC. In the context of RPC, structural conformance has also been successfully put to work, e.g., in Obliq [Car95], or Emerald [BHJ$^+$87]. In particular, it has been shown that structural conformance can be ideally combined with language interoperability. In the context of RPC, this feasibility has been illustrated through the *Renaissance* [Muc96] system and its *Lingua Franca* IDL.

TPS. The benefits of structural conformance also become apparent in the case of event-based interaction. When considering content-based queries based on method invocations, without association with an event type, the resulting filters can be viewed as promoting "pure" content-based publish/subscribe. This will become visible in the next section, which describes a simple model for content-based subscription pattern expression in TPS.

3.5 Fine-Grained Subscriptions

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 subscription patterns, in a way equivalent to content-based publish/subscribe.

3.5.1 Modelling Subscriptions

The following abstract model supports the expression of complex subscriptions in a way that preserves the encapsulation of event objects [EG01a].

Overview

Roughly speaking, the application programmer defines *conditions* on obvents, by specifying *methods* through which these objects should be queried, along with *expected values* that are compared to the values returned by invoking these methods.

This results in an entirely novel flavor of content-based subscription. At present, all approaches to content-based subscription, even when viewing events as instances of application-defined types, rely on a more primitive view of these events, i.e., as sets of attributes without any associated methods.

Accessors

Accessors are specific objects used to access partial information on the runtime event objects.

Nested Invocations. An accessor a is characterized by a set of *method-parameters* pairs $(m_1, p_1), ..., (m_v, p_v)$, where every m_i is a method and $p_i = p_{i,1}, ..., p_{i,i_l}$ its corresponding argument list. Whenever a method m_i is applied to an object, this subsumes the fact that m_i is invoked with its arguments p_i.

Applying Accessors. An accessor can also be seen as a function which, applied to an object, returns another object:

$a(object\ o) \mapsto\ object$

When applied to an event object o, the accessor's methods are applied to o in a nested way in order to obtain relevant information from that obvent. More precisely, m_1 is invoked on o and every method m_{i+1} $(0 < i < v)$ is recursively invoked on the result of m_i. Finally, the result of m_v is returned:

$a(o)\ \Leftrightarrow\ o.m_1(p_{1.1}, p_{1.2},\ ...,\ p_{1.1_l}).\ ...\ .m_v(p_{v.1}, p_{v.2},\ ...,\ p_{v.v_l})$

With $v = 0$, the obvent o itself is accessed as a whole. The special case $i_l = 0$ reflects the case where m_i is an argument-less method.

Note however that we do not consider here potential side effects of methods, e.g., the modification of the state of the corresponding objects through queries. The responsibility of avoiding such situations is left to the user.

Conditions

While an obvent is queried through an accessor, a condition evaluates the obtained information, i.e., decides whether it represents a desirable value.

Applying Conditions. A condition represents a single constraint that a subscriber puts on obvents, and can be viewed as a function taking an object as argument and returning a boolean value indicating whether the object complies with the represented constraint:

$c(object\ o) \mapsto\ boolean$

A condition makes use of an accessor a to obtain relevant information from an obvent, and compares the obtained information to an expected result r (possibly a range of values) according to a binary predicate b, which can be viewed as a function taking two objects as arguments, and returning a boolean value:

$b(object\ o,\ object\ r)\ \mapsto\ boolean$

Evaluating a condition for a given obvent o hence results in evaluating the encapsulated binary predicate for (1) the value obtained by applying the accessor to o, and (2) the expected result r:

$c(o)\ \Leftrightarrow\ b(a(o), r)$

Object Queries. Figure 3.1 schematically outlines the proposed scheme. Similar approaches can be found for object queries in object-oriented data management systems, e.g., *Tigukat* [SO95]. In fact, both publish/subscribe messaging systems including a notion of content-based filtering, as well as database systems, deal with large amounts of data. The major difference between queries in an object database and the filtering of obvents by a middleware is the *duration* of a query. With a middleware system based on content-based publish/subscribe, the query is expressed for *future* objects. In object databases, queries are ideally instantaneously performed on objects that are already in the database, i.e., a snapshot of a usually centralized database. Since a database client is blocked until the reply of the query is obtained, most commercial solutions are willing to sacrifice encapsulation to reduce overhead and thus latency.

However, the expression of a query can be made similarly in both situations. This has been nicely demonstrated by Tigukat [SO95], which has attracted our attention mainly because it provides a full encapsulation of objects in the database. These objects are always viewed as instances of abstract data types, and, similarly to our approach, queries are based on invocations on these objects.[5]

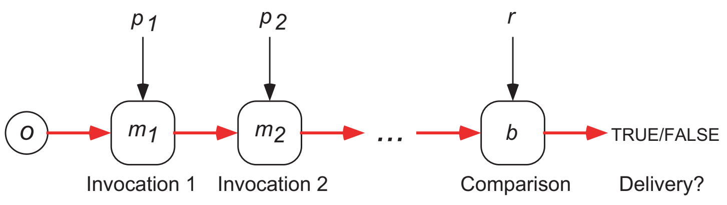


Figure 3.1: Content-Based Filtering with Method Invocations

Comparisons. Conditions vary by the comparisons they encapsulate, and many different comparisons can be imagined based on relationships between objects. Prevalent comparisons between arbitrary objects are based on the following characteristics:

Identity: In object-oriented settings, a unique identity is usually associated with every object. Two variables can hence point to distinct objects, or to the same object.

State (value): Two distinct objects can represent the same "value", or distinct "values". These "values" are usually defined by the state of the objects (or parts of it). This type of comparison is usually implied by the previous one, i.e., it is *reflexive*: an object has a same value than itself, and hence it is not important how "deep" a value-based comparison goes: two top-level objects whose attribute variables point to the same objects are equal in value. Furthermore, value-based comparison is *transitive* (if an object o_1 is equal to an object o_2 and o_2 is equal to o_3, o_1 and o_3 should be equal as well), *symmetric* (if o_1 is equal to o_2, then o_2 is equal to o_1) and *consistent* (if two objects o_1 and o_2 are equal in value at time t_0, they should be so at any t after t_0, for as long as the individual values do not change).

Ordering: An order can be defined on objects, based on different criteria. This enables more refined value-based comparisons than the previous strict equality check. Relationships are usually represented by symbols such as <, or <=. Just like the above equality, these are *consistent*, *transitive*, e.g., $o_1 < o_2$, $o_2 < o_3$ $\Rightarrow o_1 < o_3$, yet *anti-symmetric*, e.g., $o_1 < o_2 \Rightarrow o_1 \not> o_2$.

[5]Reflective mechanisms also enable a closer integration of the language with the object database [PO93].

While the latter two types of comparisons make much sense when comparing event objects, the first one is not typical for a distributed context like TPS: according to the general object model and the semantics of obvent publishing defined above, every published obvent triggers the reception of a new, distinct clone by every subscriber. Thus it does usually not make sense for a subscriber to "seek" for a precise event object: whether itself publishes that event object or someone else, the received object will in any case have a new distinct identity. This type of comparison can however become useful in special cases, e.g., when a subscription criteria involves objects *reifying* environment parameters which are viewed as singletons.

Primitive types. When thinking of these standard comparisons and operators, primitive types such as floats, bytes, or booleans immediately come to mind. When picturing events as ultimately composed of (nested) attributes of mainly such primitive types, the importance of standard comparisons becomes clear.

In object languages which are designed according to the uniform philosophy that "everything is an object", e.g., Smalltalk or Eiffel, such standard comparisons lead to corresponding methods. Also, certain languages support *operator overloading* as a means of redefining such operators for primitive object types, or for arbitrary application-defined types. C++, for instance, enables the definition of the previously mentioned operators for basically any object type.

Accessors or comparisons? One can always think of more comparisons between the result of an accessor and expected values, e.g, relationships such as $\in$ or $\subset$. An interesting example could consist in checking an object for its conformance to a given type. Many languages support this, either through a primitive (for static checks), or also by reifying types and supporting the dynamic querying of an object for its type (e.g., Java, Smalltalk). In the second case, a specific method is implemented by every object, and the returned value can be easily compared with another reified type. In the above framework, there are two things as part of which this specific method can be viewed:

Accessor: Being at the end of the invocation chain, this method can be viewed as m_v. The remaining comparison then simply consists of a state-based identity check, since reified types are usually represented as unique objects. This represents one of the rare situations where object comparisons based on identities would make sense.

Comparison: In this case, the method is viewed as part of the comparison. The comparison then becomes a specific one, in that it verifies whether the type of the object obtained through the accessor is equal to the type represented by r.

We believe that putting as much as possible into the accessor, and providing conditions for only basic comparisons, like the ones outlined above, is the more general approach. However, conditions encapsulating methods $m_u, ..., m_v$ $(u \leq v)$ can be of interest if a quicker evaluation is guaranteed, or if the expressed shortcut represents a true gain in terms of ease of use.

Subscription Patterns

Subscription patterns, akin to conditions, are evaluated against objects (obvents), returning a boolean value indicating whether the considered obvent is of interest for the corresponding subscriber:

$s(object\ o) \ \mapsto \ boolean$

Subscription patterns are however viewed as logical combinations of individual conditions (e.g., *and*, *or*). To that end, a subscription pattern s is pictured as comprising a set of w basic conditions $c_1, ..., c_w$, which are all evaluated for a given obvent o, and a function f which takes w boolean values as arguments, and returns a boolean value:

$f(boolean\ b_1,\ ...,\ boolean\ b_w) \ \mapsto \ boolean$

When a subscription pattern is evaluated, all conditions are first evaluated.[6] After that, its function f is evaluated, by combining the results of the individual conditions accordingly through f:

$s(o) \ \Leftrightarrow \ f(c_1(o),\ ...,\ c_w(o))$

3.5.2 The Never-Ending Story: Attributes vs Methods

The direct use of attributes instead of methods can be chosen as alternative to methods to describe accessors, and hence to express subscription patterns.

Politically Speaking

In general, the controversy between directly accessing attributes, or forcing the use of getter-/setter-methods to handle these attributes, is an ever recurring issue. Though everyone seems to agree that encapsulation is vital to object-oriented programming, the direct manipulation of attributes is still a widely accepted practice.

For once, this issue seems to be less a conflict between academia and industry, but between a data- and logic-oriented view. While defenders of the former view tend to reason in terms of data and their representation, and, as illustrated by the database community, do not seem always comfortable with encapsulated objects, the latter view is more inclined to objects, since these greatly simplify the development and maintenance of application logic.

Technically Speaking

Programming languages handle attribute accesses differently. Eiffel, like Cecil, offers a nice compromise by automatically generating methods accessing attributes. In Eiffel, the exact name of the attributes are used for the access methods, and since

[6]In practice, conditions are evaluated one by one, and as soon as f could not be satisfied anymore, any further evaluation is aborted.

argument-less methods can be called without trailing empty parentheses `()`, direct attribute access and attribute access methods are unified. More recently, Java, like so many other languages, does not prohibit direct attribute accesses. Instead, Java offers the possibility of synchronizing these accesses.

Beyond defending encapsulation, methods have been not chosen in the model merely in attempt to abolish any direct manipulation of attributes. Methods simply offer more expressiveness, in that an attribute access can always be modelled as an invocation of a method, while the opposite is not always possible. As a side effect, the use of methods helps contradict the wide-spread belief that object-orientation and publish/subscribe are inherently opposed in that any form of content-based publish/subscribe *necessarily* breaks encapsulation [Koe99].

3.5.3 Degrees of Expressiveness

Compared to earlier models of subscription patterns, e.g., [ASS$^+$99, CRW00], the above model for invocation-based subscription patterns offers an increased expressiveness, by getting closer to an object-oriented programming model. We discuss here several considered levels of expressiveness. Each level corresponds to an improvement over "classic" content-based subscription schemes. We present these improvements in an order reflecting their respective gains in expressiveness, starting by the most significant. The impact of increased expressiveness on performance, and how to deal with it, will become more visible in the following chapter.

Nested Invocations

By offering the possibility of performing nested invocations, an arbitrarily fine grain of matching can be achieved. In contrast, viewing events as dynamic structures with one single level is characteristic for most prevalent content-based systems. With nested invocations, events can come closer to a "general" notion of objects, and their design is less influenced by a future use with a TPS system. Now obvents can have attributes which are objects and have attributes themselves, etc., without having to project the attributes' methods to the top-level obvent type.

Invocation Arguments

The advantage of nested invocations over "flat" structures would already be offered without supporting method calls with parameters. A minimalistic approach, also avoiding direct manipulation of attributes could consist in only supporting parameterless methods for reading attributes. By allowing parameters however, expressiveness is again increased, and the application logic can be encapsulated in the obvents. Consequently, the expressiveness is less dependent from possible comparisons offered by conditions, etc.

Accessors as Expected Results

A further degree of expressiveness can be added by enforcing the use of accessors in the place of expected values for conditions: conditions can be formulated as comparisons of an invocation chain on obvents with the result(s) of one or more additional chains, e.g, the difference between the nominal value of a stock quote and its current value.

Accessors as Invocation Arguments

Moreover, accessors can be accepted as actual parameters for other accessors. This improvement presents an added value similar to the previous one. With the use of methods with arguments for describing subscription patterns, on one hand, the developer *can* use the full logic contained in an obvent type for querying. With the use of accessors as expected results or invocation arguments on the other hand, the developer *does not have to* think of possible comparisons, and include these as logic into the obvents.

Handling Exceptions

In many object-oriented programming languages, *exceptions* are supported as means of signalling the occurrence of abnormal events. Such exceptions can sometimes be raised by the runtime environment, or, on purpose by the application at an any execution point. To support the full semantics of methods invocations, subscription patterns expressed in such languages can also be equipped with the possibility of handling such exceptions occurring during the execution of a method in the context of obvent filtering. Such exceptions can for instance also be handled by defining a value which is then used like a return value of the method where the exception occurred, possibly by shortcutting the remaining nested invocations.

Limits

For quite obvious reasons, the above list is not an exhaustive enumeration of all possible levels of expressiveness that could be achieved in an object-oriented language. One could come up with additional levels, for instance by considering passing parameters in another way than by value. Accessors could then also be used as parameters to invocations, or to reflect expected results. Ultimately, an expressiveness very close to what you can do in the language itself can be achieved.

One could even think of method invocation chains rooted at other objects than the queried obvent, e.g., a proxy for RPC interaction to implement filters which, though migrated for performance reasons to remote hosts, are continuously (maybe not at every invocation, but regularly) updated by the subscriber to reflect the latest preferences.

Mobility vs Expressiveness

Carzaniga et al. [CRW00] point out a tradeoff between expressiveness and scalability, based on the observation that the more complex the subscription criteria become, the less coverage can be expected between the individual subscriptions, making optimizations hard.

With the increased expressiveness that we promote here by relying on application-defined code, an additional tradeoff is introduced between mobility of filters and their expressiveness. Indeed, filters should be defined in a way such that they become open to the middleware, in order to enable the middleware to perform optimizations by regrouping filters of several subscribers and factoring out redundancies between these.

Filters are hence potentially moved to foreign hosts, where they are applied at a more favorable stage. The more complex such filters become, the more difficult it is to guarantee that they do not rely on any local resource or bound object, and to foresee their effects in general. Consider only the complexity introduced by making use of subtyping: if an actual argument provided for an accessor is an instance of a subtype of the corresponding formal argument, the corresponding class might have to be transferred to the filtering host. (We will come back to the feasibility of such expressiveness levels in Section 5.)

3.6 Ensuring Type Safety

Now that we have discussed a model of subscription expression which enforces encapsulation, and discussed "what to subscribe to", the main remaining issues concern "how to subscribe", and "how to ensure that you get what you subscribe to". While the second issue will be covered in the next two chapters, and the third one partially depends on it, we give below an overview on how to deal with the last issue in a way that ensures type safety.

3.6.1 Sources of Type Errors

In the context of an implementation of type-based publish/subscribe, we roughly distinguish two places where type errors can occur.

Obvent dissemination: It has to be ensured that published obvents are conveyed correctly by the TPS engine. That is, obvents are routed to all subscribers whose subscription criteria match these obvents. Besides requiring an adequate subscription mechanism (see next chapters), avoiding errors at this stage mandates a sound implementation of the TPS engine, and correctly defined and applied content filters.

Obvent delivery: Furthermore, it has to be ensured that a subscriber effectively "receives" obvents of the correct type. A satisfactory behavior with respect to this requirement depends on the way obvents are delivered from the engine to the subscriber, relying more on the promoted abstractions.

3.6.2 Type Safety in Obvent Dissemination

A key for achieving type safety in obvent dissemination is type safety in subscription patterns, i.e., filters. If type information is available, type checks can be enforced to verify that a method m_1 specified by a subscriber for filtering is indeed defined in the subscribed type T, and recursively, every m_i is defined on the return type of m_{i-1}. Also, the types of the actual arguments $p_i = p_{i,1}, ..., p_{i,i_l}$ can be verified for their conformance with the formal arguments of m_i. These type checks can be performed at runtime by the TPS middleware, or, when expressing filters through the programming language itself, even at compilation.

3.6.3 Type Safety in Obvent Delivery

While approaches like JavaSpaces ensure that any obvent delivered to a subscriber conforms to the subscribed type (given implicitly by the type of the template object), the interaction between the JavaSpace and the callback object representing the subscriber is untyped, requiring a cast in the application code.

The Goal

In a statically typed language, such an interaction should take place in a statically type-safe manner. When subscribing to a type T, whether implicitly or explicitly, it should be ensured that the received instances are of type T, and are at least delivered as instances of type T, i.e., are assigned to variables of type T. Using any supertype of T can lead to errors when performing type casts. Returning a more narrow type is obviously only possible if the subscriber advertises the expected type, i.e., offers variables of subtypes of T.

Interaction which satisfies our criteria of type safety can be achieved in several ways. We give a preview of the two alternatives dealt with in the two following chapters (in reverse order).

Generating Typed Adapters

The most straightforward way seems to be the generation of some form of typed proxies. In [OPSS93] these are termed *adapters*, which helps avoiding confusion; the term proxy is prevalently used in the context of RPCs, where it denotes an object which mimics the remote object [Sha86], and thus provides an interface which conforms to the interface of that remote object. In the case of TPS, adapters provide interfaces which are *similar* among each other: they differ in the types of formal arguments representing obvents. This type is accorded to the type of obvents they are used with.

Similarly to proxies however, adapters are intermediate entities between the communication system and the application, whose role consists mainly in mediating between serialized data and a more higher-level representation.

Such adapters can be generated by a specific (pre)compiler. As shown by second class RPC packages, a precompiler offers the ideal way of dealing with interoperability. This scheme will be applied in our language integration approach described in Chapter 5.

Type-Parameterized Adapters

As we will illustrate in the next chapter, adapters can also be viewed as proxies for a ubiquitous first-class channel abstraction. In such a case, a more dynamic solution than above can be obtained, which is strongly encouraged by languages incorporating reflective capabilities like Java or Smalltalk, where basically anything can be done dynamically (with a corresponding cost in performance).

To avoid type errors at obvent delivery, the same adapter type can be parameterized by the obvent type. Thanks to reflective mechanisms, the behavior of these adapters can be adapted at runtime to the types used, avoiding type errors at obvent dissemination.

3.7 Discussion

In this section, we discuss several issues related to TPS. In particular, we investigate more closely the limits between TPS and content-based and subject-based publish/subscribe, as well as related approaches to typed publish/subscribe.

3.7.1 Type-Based vs "Traditional" Publish/Subscribe

We compare here TPS with conventional publish/subscribe styles, pointing out the fact that TPS can be viewed as a higher-level variant of publish/subscribe, which nevertheless allows the expression of its ancestors, i.e., the subject-based and content-based variants.

The State is the Content

With TPS, similarly to content-based publish/subscribe, publishers have no knowledge of any "address" they are publishing to. The set of subscribers which potentially receive a given obvent is implicitly defined by the set of subscribers whose criteria match the "content" inherently given by the state of that obvent. Nevertheless, subscription patterns are not systematically described through this state, and equally, this state is not necessarily accessed directly upon filtering.

A content-based scheme in the sense advocated by most systems implementing a form of content-based subscribing, i.e., based on some form of self-describing events

(Figure 2.11), can be easily achieved with TPS. Since any event type can be used, nothing prevents from using a single event type representing dynamic structures.

By furthermore describing subscription patterns based on the methods provided by such a type, encapsulation can be effectively preserved.

The Type is the Subject

It has been widely recognized that content-based publish/subscribe is more general than its subject-based kin. A subject-based scheme can be easily deployed on top of a content-based engine, by associating an attribute representing a set of subject names (reflecting a hierarchy), e.g., an array of character strings, with every event.

A TPS engine can be used straightforwardly to express the traditional subject-based publish/subscribe. By introducing a root obvent type with a method to read a subject attribute, one can easily perform subject-based matching on obvents. As depicted in Figure 3.2, this can be done in a way that promotes subjects as static in nature, along with dynamic content-based publish/subscribe.

Note however that we do not consider performance issues here. The amount of static information introduced in subject-based systems, by allowing subjects to be predefined (e.g., set up by a system programmer), can be exploited to achieve considerable speedups. In comparison, systems which are completely dynamic, e.g., pure content-based systems, or subject-based systems where subjects can be defined arbitrarily, are more difficult to implement efficiently.

```
public class SubjectEvent extends DynamicallyStructuredEvent {
  public static final String wildcard = "*";
  private String[] subject;
  public String[] getSubject() { return subject; }
  public boolean matches(String[] isOfSubject) {
    if (isOfSubject.length > subject.length) return false;
    for (int i = 0; i < isOfSubject.length; i++)
      /* verify if subject name component is same, or wildcard */
      if (isOfSubject[i] != subject[i] && isOfSubject[i] != wildcard)
        return false;
    return true;
  }
}
```

Figure 3.2: Expressing a Mixed Subject/Content-Based Scheme with TPS

TPS and RPC

Clearly, the goal of supporting TPS is not to replace any other distributed programming abstraction. In fact, all abstractions outlined in the previous chapter have proven their superiority in certain contexts, while being quite inefficient in others.

In particular, the RPC has been often claimed to be *the* interaction paradigm for distributed object settings. There are mainly two differences between such remote invocations and our obvent-based model:

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

Object passing semantics: With remote invocations, there are two possible ways of passing objects, namely (1) by *reference*, i.e., a proxy object is created in the receiver's address space, and (2) by *value*. The distinction goes along the notion of the granularity mediated in Section 3.2, i.e., "large" locality-bound objects interact via remote invocations, where the invocation arguments are "small" unbound objects or references to other "large" remotely invocable objects. In contrast, when using our obvent-based model, objects are primarily passed by value.

However, TPS and RPC are not contradictory, but complement each other. A combination of both represents a very powerful tool for devising distributed applications, e.g., by passing object references as, or with, obvents. This will be illustrated in the next two chapters, through an identical example implemented according to the two approaches outlined in those two chapters.

3.7.2 Related Notions of Typed Publish/Subscribe

Several similar, yet not equivalent, notions of typed publish/subscribe have been described in the literature. We discuss these here mainly with respect to type safety and encapsulation, pointing out the fact that they altogether fail in providing both type safety with application-defined events (Table 3.1), and content-based queries on these preserving encapsulation (Table 3.2).

COM+

Microsoft's COM+ [Obe00] promotes a model based on the types of subscribers: the application can provide a specific interface defining its own operations through which it will be called. Because the provided event notification service supports asynchronous (oneway) interaction, operations are not allowed to return results. With this incarnation of the oneway proxy abstraction known from RPC (see Section 2.2.4), events are hence represented by asynchronous invocations, but are not reified.

The primary filtering is thus made on the types of the subscribers, leading to a more explicit notion of destination than in the case of TPS. Though subscribers are not specified by their identities, they are specified by their type. In contrast, TPS promotes the implicit type of events as notion of destination.

By viewing an invocation as an event, the invocation arguments can be viewed as the attributes of the resulting notification. Filters in COM+ are based on *argument-value pairs*, i.e., expected values for invocation arguments, and are expressed through a limited subscription grammar. Encapsulation seems to be preserved by avoiding the reification of events. To ensure type safety, the application is responsible for providing a typed "dummy" proxy (an `EventClass` object), that will only be used during compilation.

CORBA Event Service

As already mentioned, the OMG has specified its own CORBA service for communication based on the publish/subscribe paradigm, known as the CORBA Event Service [OMG01b]. According to the general service specified, a consumer interacts with an *event channel* expressing thereby an interest in receiving *all* the events from the channel. In other words, the filtering of events is done according to the channel names, which basically correspond to subject names. However, no containment relationships are implicitly defined for subjects, that is, there is no inherent support for hierarchies.

A form of typed interaction is provided, similar to the model in COM+, enabling the use of the types of consumers *or* producers, since the event service supports pull- and push-style interaction. All operation parameters in the former case must be tagged `out`, and in the latter case `in`. Typed proxies are generated based on the application's interface, which in practice requires a specific compiler. According to [OMG00], the specification for typed interaction is difficult to understand for many users, and implementors find it hard to deal with. Most implementations therefore only provide untyped events.

TAO Event Service. The deficiencies of the CORBA Event Service, such as the difficulties with typed events as well as missing support for QoS and realtime requirements, were apparent soon after commercial implementations became available. They have been well documented by Schmidt and Vinoski [SV97].

The former author was involved in the development of the *TAO Realtime ORB*, which brought up the probably most significant service implementation incorporating proprietary extensions ([HLS97]). Together with other extended event services (e.g., *OrbixTalk* [ION96]), the TAO Realtime ORB plausibly demonstrated the deficiencies of the CORBA Event Service specification, emphasizing mainly realtime and QoS issues. Nevertheless, events are still viewed as structures, and filtering is based on the identities of publishers and/or the event types. In the latter case, the "type" is represented by the value of an attribute of enumerative type, and does hence not affect the interfaces of the service.

CORBA Notification Service

After the emergence of such extended and proprietary approaches aimed at fixing the shortcomings of the event service, the OMG issued a request for proposal for an augmented specification. The outcome, the CORBA Notification Service [OMG00], promotes the *notification channel* as an event channel with additional functionalities. Notions like priority and reliability are explicitly dealt with.

A new form of (semi-) typed events, called *structured events* is introduced. These represent a form of dynamic structures, which are roughly composed of an event header and an event body. Both parts consist of a fixed part. The fixed header for instance consists of attributes like event type and event name. The fixed body can carry anything (reflected by the CORBA `any` type). The specification also describes conversions between different kinds of events, e.g., between "typed" and structured events.

The variable parts of structured events are composed of name-value (name-`any`) pairs, for which the specification mentions a set of standardized and domain-specific mappings, while most applications will require their own mappings. In the context of content-based filtering, the name-value pairs are seen as the attributes of the event and are directly accessed through *filters*. Constraints are described as strings following a complex subscription grammar given by the Default Filter Constraint Language, which extends the OMG's *Trader Constraint Language*. A second type of filters, called *mapping filters*, filter events with respect to the desired QoS.

JavaSpaces

Inspired by the Linda tuple space [Gel85], a JavaSpace [FHA99] is a container of objects that can be shared among various suppliers and consumers. The JavaSpace type is described by a set of operations among which a `read()` operation to get a copy of an object from a JavaSpace, and a `notify()` operation aimed at registering a consumer object which is to be alerted about the presence of some specific objects in the JavaSpace. With a JavaSpace, one can thus build a publish/subscribe communication scheme in which the JavaSpace plays the role of the event channel aimed at broadcasting event notifications to a set of subscriber objects. Custom events can be defined by subtyping the basic `Event` type, which is used for formal arguments representing events in the API. Type safety is hence not ensured, and it is unavoidable that nasty type checks and type casts pollute the code.

A given subscriber of a JavaSpace advertises the type of events it is interested in by providing a template object t. A necessary condition for o, an object notifying an event, to be delivered to that subscriber, is that o conforms to the type of t. Furthermore, the attributes of t are compared byte-wise with the corresponding attributes of o, with `null` playing the role of wildcard. Besides violating encapsulation, this scheme suffers from the severe limitations in expressiveness of template-based subscription schemes already outlined in Section 3.1.2. In the case of JavaSpaces, moreover, attributes cannot be of primitive types, must be declared public, and since `null` is reserved for the sake of representing an arbitrary value, it cannot be used as matching criterion.

ECO

An approach to integrating event-based interaction with a programming language, namely C++, is discussed in the ECO (*events + constraints + objects*) model [HMN$^+$00]. The authors also dissociate the two main ways of adding event semantics to an object-oriented language, opting however for the second one; the addition of events as specific language constructs decoupled from the main application objects. Their ECO model hence necessitates a considerable number of language add-ons. The use of a precompiler to handle these extensions is also mentioned, but [HMN$^+$00] gives no details about its implementation. Filtering can be based on the publisher's identity (the source), and several types of *constraints. Notify constraints* are expressed based on the *parameters* (attributes) of events, once more neglecting encapsulation by inherently representing events as sets of attributes. Further constraints, like *preconstraints* and *postconstraints* use the state of the receiving instance. All constraints, as well as methods for event handling are declared on a per-class base, while subscriptions/unsubscriptions, i.e., contracts between the event handlers and the handled event types, are performed instance-wise. The potential mismatches implied by this will be discussed more in detail in Section 5.5.2.

CEA

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

3.7.3 Object Model

We discuss the object model presented in Section 3.2 in detail, focusing on its limitations and certain alternatives.

Publishers

In the above model we do not introduce a specific publisher type. In fact, most variants and applications of the publish/subscribe paradigm inherit the property of "anonymity of participants" from the tuple space paradigm, which is viewed as a

System	Type of	Reflected by	Implementation
COM+	Subscribers	Typed Proxy	Appl.-Provided Proxies
CORBA Event S.	Subscribers	Typed Proxy	Precompilation
CORBA Notif. S.	Subscribers Events	Typed Proxy String attribute	Precompilation Dynamic Structures
TAO Event S.	Events	Integer Attribute	Dynamic Structures
JavaSpaces	Events	Inherent Type	Untyped Proxies
ECO	Events	Inherent Type	Precompilation
CEA	Events	Inherent Type	Precompilation

Table 3.1: Common Approaches to "Typed" Publish/Subscribe

System	Type of	Content Matching
COM+	Subscribers	Argument Aalues
CORBA Event Service	Subscribers	No Support
CORBA Notification Service	Subscribers Events	Attribute Values Attribute Values
TAO Event Service	Events	No Support
JavaSpaces	Events	Attribute values
ECO	Events	Attribute Values
CEA	Events	Attribute Values

Table 3.2: Content-Based Support in "Typed" Publish/Subscribe

merit (*anonymous communication* [OPSS93]). Making the *type*, or even *identity*, or a publisher visible to subscribers is useful in some cases, and can still be achieved without viewing publishers as specific objects. Using the publisher type or identify as a subscription criterion, as successfully promoted by some approaches overviewed in the previous section, such as the TAO Event Service, or the original CORBA Event Service, in the case of typed pull-style interaction,[7] leads to interesting solutions. Nevertheless, we focus here on the types of the exchanged events, and on a complete anonymity of participants at the abstraction level.

By not considering publishers as specific objects, any object can publish obvents. In particular, obvents can publish obvents, even themselves.

Subscriptions

Subscriptions are similarly viewed as actions involving subscribers. A subscription is however not reflected in the state of the corresponding subscriber. In particular, a subscription cannot be handed over by simply transferring the corresponding subscriber object. To emphasize the generally local nature of a subscription, subscribers

[7]Note that such an interaction style does not correspond exactly to the definition of the publish/subscribe paradigm given in Section 2.4, which relies on a more push-style interaction.

have been defined as bound objects above. The opposite would have required a considerable effort to avoid all possible conflicts. In particular, obvents could have been given the possibility to subscribe to obvents, which however might lead to non-trivial issues.

This becomes visible when considering an obvent o which is published, yet is a subscriber itself. What about obvents corresponding to o's subscription criteria which are published between the moment when the publication of o is triggered and the moment when a considered copy of o is created for a given subscriber? In fact, if the obvent notified to any subscriber whose criteria correspond to o is to be an exact copy of the original obvent o, one could argue that any copy o' of o must receive all obvents received by o after o has been published, since that is the moment when the creation of o' has been triggered. This would require a considerable effort at the implementation level, to replay obvents published meanwhile. Given this assumption moreover, the case when o subscribes to itself (*self-subscription*) would lead to an endless recursion.

Alternatively, one could consider the publication of an obvent as an atomic action, in the sense that a copy of o only receives obvents published after its creation. This would however impose strong synchronization constraints on the system.

Summary

TPS is to publish/subscribe what remote method invocations are to RPC: a higher-level, object-driven paradigm variant. TPS strives for an “easy” programming model. In our case, we have identified mainly type safety and encapsulation as key concepts for assisting developers in safely devising distributed applications. Achieving these without reducing the high expressiveness inherited from content-based publish/subscribe introduces new challenges, e.g., how to express type-safe subscription patterns which are nevertheless transparent to the underlying TPS middleware.

TPS is nevertheless general, in the sense that it captures traditional forms of publish/subscribe. In particular, TPS can be used to express more traditional flavors of publish/subscribe, like the content-based variant. TPS can hence be put to work in a first-class TPS package, i.e., supporting a single language like Java, as illustrated in the following chapters, or could also be imagined in a language-interoperable second class TPS package.

Chapter 4

GDACs: A TPS Library

Our first implementation of TPS is based on a library approach, which provides a first-class channel abstraction to the application developer. Channels correspond to obvent types, i.e., a channel representing type T can be used to publish obvents of type T (including any subtypes of T), and to issue subscriptions to type T.

These channels are viewed as instances of a particular variant of the well-known *collection* abstraction. These *Generic Distributed Asynchronous Collections* (GDACs) are collections specialized for distributed and asynchronous interaction based on publish/subscribe [EGS00, EG00, EG01a, EGS01]. According to the commonly chosen approach of providing families of different collections addressing various purposes in *collection frameworks*, we propose different GDAC (sub)types, reflecting different QoS.

GDACs are implemented in GJ, an extension of Java adding *genericity*, and fit naturally into the collection framework offered as part of the Java environment. GDACs constitute one of the foundations of the *Distributed Asynchronous Computing Environment* (DACE) [DACE], a project targeted at exploring algorithms and abstractions for large scale distributed programming.

4.1 Background: Collections

Before introducing our Generic Distributed Asynchronous Collections (GDACs), we first roughly overview collection frameworks in object-oriented programming languages.

4.1.1 Language-Integrated Collection Frameworks

A collection framework is a unified architecture for representing containers for objects, allowing them to be manipulated independently of their representation. Such collection frameworks are offered by many object-oriented languages as part of inherent class libraries.

Smalltalk

In a language like Smalltalk, in which everything is an object, and even control structures like loops are provided by classes, it seems natural to provide a rich set of libraries as part of the language. In particular, Smalltalk comes with a collection framework, rooted at the `Collection` class, that reduces the programming effort by providing useful data structures and algorithms together with high-performance implementations ([IBM95]).

Java

Java follows this model by providing a package `java.util` [Sun00c] encompassing various kinds of object containers, like *sets* or *hashtables*, as part of its environment [Sun00b]. Figure 4.1 shows an extract of the type hierarchy of the Java collection framework.

Despite the various purposes addressed by the inherent Java collection framework, many additions have been proposed, like the *Java Generic Library* (JGL) [Obj99] which leverages remotely accessible collections, or the collections proposed by Lea [Lea97a] for more fine-grained synchronization between accessing objects.

4.1.2 External Frameworks

Unlike Java or Smalltalk, which provide rich sets of class libraries, several languages have been initially conceived without standard libraries in mind, and thus lack such a collection framework. For C++, this lack has been compensated by the *Standard Template Library* (STL) collection framework [SL95]. The STL is probably at the present one of the most complete and widely employed library of collections.

Note that the fact that Eiffel has failed in bringing out a set of standard libraries is often stated as reason for the language's stifled spread.[1] In particular, many developers find Eiffel's lack for an inherent collection framework very inconvenient and discouraging.

4.2 GDACs Overview

Just like any collection, a GDAC is an abstraction of a container object that represents a group of objects. It can be seen as a means to store, retrieve and manipulate objects that form a natural group, like a mail folder or a file directory.

[1]Besides, of course, the obvious problems induced by the way its type system implements *covariant subtyping* [Coo89], and fundamental marketing and opportunistic issues.

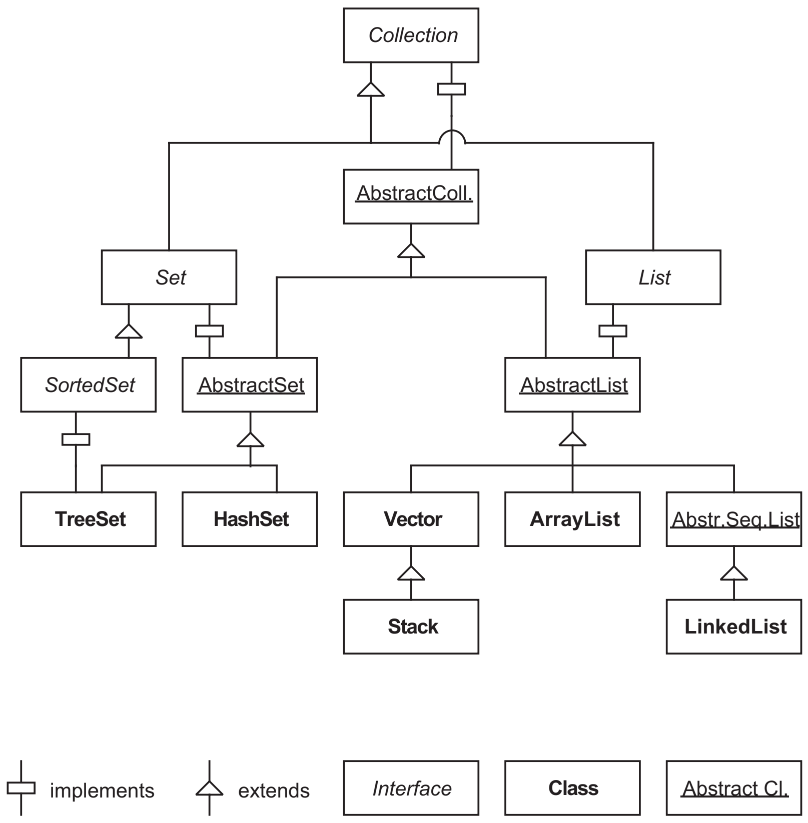


Figure 4.1: Collections in Java (Excerpt)

4.2.1 "G" for "Generic"

By interacting with a given GDAC, one expresses interest in obvents of a particular kind, namely those handled by that GDAC. Since in TPS the notion of *event kind* is equivalent to *event type*, TPS can be straightforwardly expressed through GDACs for specific types. If a GDAC corresponds to a type T, any obvent conforming to type T can be published through that GDAC, and subscribers can express their interest in obvents of type T through that GDAC.

Background: Genericity

Since all obvents entering and leaving a GDAC for a type T are of that same type T, T should appear as the type of every formal argument representing an obvent in the interface of the GDAC. Strongly typed interaction can be achieved without generating specific interfaces nor classes for every used obvent type by relying on *genericity* [Mil77], where the obvent *type* handled by a GDAC is viewed as a *parameter* or *attribute* of the GDAC.

Parametric types. One face of genericity is usually referred to as *parametric polymorphism*, and the resulting types are called *parametric types* alluding to their *type parameters*. Forms have appeared as *template*s in C++ (*unbounded* parametric polymorphism), in Eiffel [Mey86] (*bounded* parametric polymorphism: bounds can be defined on type parameters, like formal arguments), and Sather (*F-bounded parametric polymorphism* [CCH$^+$89]: a type parameter can appear as parameter of its bound). The latter form underlies most of today's efforts, and has turned out to be particularly convenient for *binary methods* [BCC$^+$95].

Virtual types and other forms of genericity. Another variant of genericity consists in representing such a type parameter as an effective attribute of the generic type, which provides useful runtime information. Such types are commonly called *virtual types*, and have been initially introduced in the BETA language [KMMPN83]. Virtual types and parametric types have been the subject of many comparisons [BOW98, TT99].

A very similar form of genericity has been adopted in Ada 95 through its `generic` types, which can be parameterized by types, but also functions or values. The possibility of extending types passed as parameters to generic types, provides for *mixin inheritance* [BC90].

Genericity in Java

Statically typed languages like Java or Oberon [Rei91], or dynamically typed languages like Smalltalk have been initially designed to support generics by the idiom of replacing variable types by the root of the type hierarchy. For such languages lacking generic types and methods, extensions have been widely studied.

Proposals. In the case of Java, there has been a true proliferation of prototypes augmenting Java with some form of genericity: Pizza [OW97], and its follow-up Generic Java (GJ) [BOSW98] both provide F-bounded parametric polymorphism, and are implemented as *homogenous* translations, i.e., type parameters are replaced by the root object type and casts are automatically inserted wherever required. Consequently, an extended compiler can do the whole work. NextGen [CS98] is a superset of GJ, inspired by Eiffel, which relies on a *heterogenous* translation (specific variants of parametric types are generated for every appearing combination of values for type parameters). Another heterogenous translation using a specialized loading of classes in the virtual machine, and including a proposal for the extension of the byte code format is given by [AFM97]. PolyJ [MBL97] provides a very flexible model through *where clauses* (also promoting a form of structural conformance) and comes with an extended virtual machine, and [Tho97] describes a homogenous translation of virtual types.

The outcome. All these efforts have proven the strong interest for some form of genericity in the Java language. As a consequence, Sun has started investigating feasible extensions to the Java language. The current prototype is based on GJ, and a first official appearance in Java is expected in 1.5.

Interestingly, the usefulness of such parametric types has been often demonstrated on collections.[2] It seems thus very promising to apply genericity to our GDACs as well. Currently, the implementation of GDACs relies on Sun's prototype, i.e., GJ.

4.2.2 "D" for "Distributed"

Unlike a conventional collection, a GDAC is however a distributed collection whose operations might be invoked from various nodes of a network. GDACs differ fundamentally from the distributed collections in JGL for instance, by being asynchronous and *essentially distributed*, i.e., GDACs can be seen as omnipresent entities. In contrast, the distributed collections in JGL are centralized collections that can be remotely accessed through Java RMI.

Participants act with a GDAC through a local proxy, which is viewed as a local collection and hides the distribution of the GDAC, as outlined in Figure 4.2. GDACs are not centralized on a single host, in order to guarantee their availability despite certain failures. We will illustrate a possible implementation in Chapter 6.

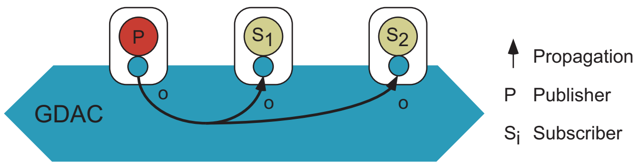


Figure 4.2: GDAC Distribution

4.2.3 "A" for "Asynchronous"

Our notion of Distributed Asynchronous Collection represents more than just a distributed collection. In fact, a synchronous invocation of a distant object can involve a considerable latency, hardly comparable with that of a local one. In contrast, asynchronous interaction is enforced with our collections. By calling an operation of a GDAC, one can express an interest in future elements. Therefore, when such an object is *eventually* "pushed" into the GDAC, the interested party is asynchronously notified.

GDACs vs Futures

The interaction between subscribers and GDACs bears a strong resemblance with the notion of future discussed in Section 2.2.4.

[2]Even in the case of C++, a good proof of the advantages of parametric types is given by the widely used STL collection framework.

Figure 4.3 compares the two paradigms. When programming with GDACs, the subscriber can be viewed as the client. The GDAC incarnates a server role in this scenario, since the publishers, which are the effective information suppliers, remain anonymous.

By calling an operation on a GDAC, the caller requests certain information. The main difference between GDACs and futures lies in the number of times that information is supplied to the client. Within the notion of future, a single reply is provided to the client,[3] whereas with GDACs, every time an information which is interesting for the registered client is created, it will be sent to it.

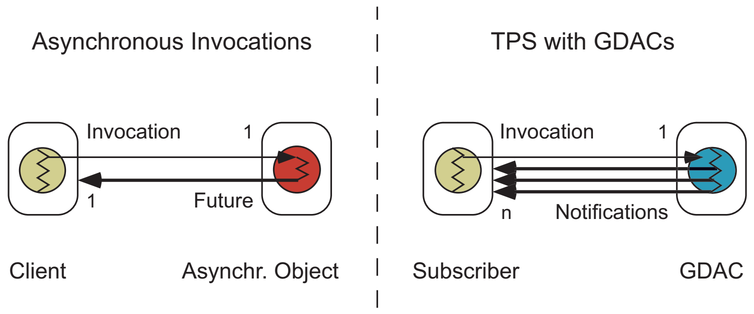


Figure 4.3: GDACs vs Futures

GDACs vs Observer Design Pattern

The interaction between subscribers and GDACs appears like an interaction based on the observer design pattern ([GHJV95], cf. Section 2.2.5). Indeed, from the perspective of a subscriber, the GDAC is not distinguishable from an effective publisher of obvents, and when subscribing, the subscriber has the impression of interacting directly with "the" publisher. According to the terminology adopted in the observer design pattern, the GDAC is hence the *subject* and its client is the *observer*.

TPS differs from a distributed implementation of the observer design pattern in that producers and consumers remain anonymous, since consumers do not directly interact with producers. GDACs, just like any channel abstractions represent neutral intermediate entities which decouple obvent producers and consumers.

4.2.4 "C" for "Collections"

As discussed already in Section 2.4 and Section 3.7.2, there are various interpretations of the publish/subscribe paradigm. In particular, the CORBA Event Service supports both pull- or push-style interaction between the channel and producers with either pull- or push-style interaction between the channel and the consumers.

Also, any form of distributed communication relying on an intermediate abstraction, e.g., shared space, comes close to the channel abstraction, and much effort

[3]Although there are exceptions, like in ABCL/1 [YBS86], where several replies may be returned.

has been made in the context of GDACs to analyze relationships between different interaction styles: in the perspective of unifying several existing interaction styles without blurring their differences, we have attempted to identify an abstraction that is both general but also supports effective implementations. This is the rationale underlying the use of the intuitive collection abstraction, which can indeed be used to unify different interaction styles.

Push

The "classic" publish/subscribe interaction style is based on push-style interaction between publishers and the channel, and similarly between the channel and subscribers. This interaction style indeed supports scalable implementations, since events are sent from the source to the destinations, and are "immediately" delivered, i.e., buffering is minimal, and the inherent all-of-n semantics remove any need for concurrency control between subscribers. This is the main interaction style with GDACs.

Pull

Nevertheless, GDACs also support pull-style interaction, however only with respect to subscribers. In other terms, obvents are always pushed from publishers towards GDACs, but consumers can choose to pull obvents from GDACs. Such a pull-style interaction can be blocking (the pulling object is blocked until an obvent corresponding to its criteria is received) or non-blocking (*polling*: if no event is available, the call returns immediately or after a given timeout). However, the same dissemination model than above (all-of-n) is provided, i.e., a given obvent is usually not only received by a single consumer.

In Section 4.7 we will illustrate the flexibility of the GDAC abstraction, by illustrating how it permits the expression of alternative interaction styles, like message queuing models, i.e., pull-style interaction of consumers with the channel in combination with a one-of-n dissemination model.

4.3 GDAC API

In this section, we present the main functionalities of our Java implementation of GDACs, reflected through their interface.

4.3.1 Methods

Figure 4.4 summarizes the main methods of the `GDAC` interface, and Figure 4.5 overviews further types in the GDAC framework. We roughly distinguish between original (adapted) methods, and specifically added methods. (More methods will be introduced later on in this dissertation.)

```
package GDACs;

import java.util.*;

public interface GDAC<T> extends Collection<T> {
  /* original methods */
  public boolean contains(T t);
  public boolean add(T t);
  /* all-of-n, push-style */
  public void contains(Subscriber<T> ts, Condition c);
  /* all-of-n, pull-style */
  public void get(ExceptionHandler h, Condition c);
  public T get(ExceptionHandler h, long timeout);
  /* consumer leaving */
  public void clear(ExceptionHandler h);
  ...
}
```

Figure 4.4: GDAC Interface (Excerpt)

```
package GDACs;

import java.util.*;

/* basic exception */
public abstract class NotificationException extends Exception {...}

/* callback interfaces for consumers */
public interface ExceptionHandler {
  public void handleException(NotificationException ne);
}
public interface Subscriber<T> extends ExceptionHandler {
  public void notify(T t);
}

/* basic condition */
public interface Condition implements Serializable {
  public boolean conforms(Object o);
  public Condition and(Condition c);
  public Condition or(Condition c);
  public Condition nand(Condition c);
  public Condition nor(Condition c);
  public Condition xor(Condition c);
  public Condition not();
}
```

Figure 4.5: Various Types used with GDACs

Adapted Methods

Since a GDAC is in the first place a collection, the `GDAC` interface inherits from the standard `java.util.Collection` interface.[4] The inherited methods are not denatured but adapted.

`contains(T t)`: A GDAC is first of all a representation of a collection of elements. This method allows to query the collection for the presence of an object. Note that an object that is contained in a GDAC is of the type represented by that GDAC.

`add(T t)`: This method allows to add an object to the collection. The corresponding meaning for a GDAC is straightforward: it allows to publish an obvent of the type represented by that collection. An asynchronous variant of this method could consist in advertising the *eventual production* of notifications. By combining with the registration of a callback object, that the GDAC would poll in order to obtain new obvent notifications from the producer, a producer-side pull-style interaction (*pullsupplier* in the terminology adopted in CORBA [OMG01b, OMG00]) could be achieved.

Added Methods

We have added several methods to express the decoupled nature of publish/subscribe interaction specific to GDACs. Not all operations known from conventional collections find an analogous meaning in an asynchronous distributed context.

`contains(Subscriber<T> ts, Condition c)`: The effect of invoking this method is not to check if the collection already contains an object revealing certain characteristics, but is to manifest an interest in any such object, that should be eventually pushed into the collection. The interested party advertises its interest by providing a reference to an object implementing the `Subscriber` interface (Figure 4.5), through which it will be notified of obvents, and an instance of `Condition`, which represents a condition according to the model represented for content-based query description in the previous chapter.

`get(ExceptionHandler h, Condition c)`: The original `get()` method found in the Java `Map` interface enables the retrieving of an element from the collection based on its associated key. In the case of GDACs, this method is reused to implement a pull model supporting several parallel pulling clients. As elucidated above, pulling is supported with all-of-n semantics. To distinguish which elements have already been delivered to which consumers, push-style consumers first register through this method with an `ExceptionHandler`, which is then used to identify consumers upon every subsequent attempt to pull a new obvent. Also, when registering, a subscription pattern is specified.

[4]Note that when we refer to the standard Java collection root type, we mean the generic variant provided as part of Sun's generic compiler.

`get(ExceptionHandler h)`: This method is effectively used to pull new elements corresponding to the criteria registered with the corresponding consumer.

`clear(ExceptionHandler h)`: The conventional argument-less `clear()` method allows to erase all elements from the collection, whereas this asynchronous variant expresses the end of a consumer's interaction with the GDAC. We have refrained from reusing the `remove()` method defined in the original collection interface, for the simple reason that we have reused that method to express destructive reading (one-of-n), leading to a form of message queuing interaction (cf. Section 4.7).

Obvent Types

While existing publish/subscribe frameworks introduce specialized event (message) types, e.g., JMS, our TPS approach frees the application programmer from the burden of explicitly inserting objects into and extracting them from instances of predefined event types. In our context, an event can be basically of any object type. In other terms, any type can be passed as value for the type parameter T, whether interface or class. By all evidence however, any candidate obvent type has to implement `Serializable` to benefit from Java's default serialization.

Introducing bounds. With GJ, one could easily express that all obvent types must be serializable, by introducing the root serializable type as bound for the type parameter:

```
public interface GDAC<T implements Serializable> {...}
```

This makes use of bounded parametric polymorphism, which is a subset of F-bounded parametric polymorphism offered by GJ, as described before-hand, which ensures at compilation that all obvent types used with GDACs are effectively serializable.

In the present library approach, to follow Java's philosophy consisting in faking a serializable root type (Section 3.3.3), and also to emphasize the fact that obvents *are* application-defined objects, we do however not impose any bound on obvent types, as shown by the `GDAC` interface.

Predefined obvent types. In contrast to specifications like JMS, which aim at wrapping industrial strength solutions for publish/subscribe and message queuing interaction, our GDACs represent a prototype, aimed at proving the feasibility of TPS. As a consequence, our APIs have been designed for intuitive use, and are certainly not as elaborate as the industrial systems which make QoS an integral part of their programming models.

As result of the testing of our GDACs in industrial contexts nevertheless, e.g., [Ros01], a set of predefined obvent types have emerged, manifesting attributes like the identity of the sender, priorities, time-to-live, etc., which seem to be of a considerable usefulness in industrial applications. This does in no way contradict one of the main driving forces behind TPS, which is the possibility of defining own types. This feature is still fully supported by giving the programmers the possibility to define their own obvent types, for instance as subtypes of predefined ones, and, unlike in approaches like JavaSpaces, type safety is ensured with any such subtypes through genericity.

4.3.2 GDACs with GJ

GJ enables the use of the original Java virtual machine, and comes as an extended Java compiler, yet fully compatible with the Sun release.

Type Erasing in GJ

As already briefly mentioned, GJ applies a homogenous translation, consisting in translating generic constructs to non-generic ones. GJ proceeds by erasing all type parameters, mapping type variables to their bounds, and inserting casts into the compiled class code. No specific classes or interfaces are thus created, as in the case of heterogenous translations. Type conversions do not take place in a generically defined class, but in the objects which invoke instances of that class. This is schematically outlined in Figure 4.6, which depicts the case of a GDAC delivering an obvent to a subscriber.

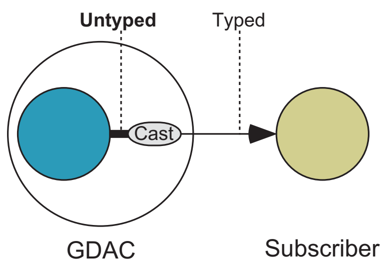


Figure 4.6: Type Erasing: GDAC Invoking a Subscriber

Runtime Support

GJ presents certain limitations, detailed in [BOSW98]. For our application purpose however, mainly its lack for *runtime support* represents a sensible shortcoming. Indeed, in our distributed context, runtime support is rather important: roughly spoken, a GDAC for a given type T has to know that its type parameter is T in order to "connect" to other GDACs for T (Figure 4.2). Note that this same issue has been pointed out as a flaw when dealing with *orthogonal persistence* [SA98]: an instance of a parametric type whose state is recovered must know its bounds to be correctly reincarnated.

Providing type information. We have considered two ways of providing an instance of a parametric type in GJ with runtime type information concerning its parameters, namely *implicitly* and *explicitly.* In the first case, information on type parameters is implicitly provided to a parametric object through an object (or an array) of the corresponding parameter type, ideally through a constructor.[5] This solution is however not fully satisfactory, since an object (similarly an array of objects) of any type $T2 \leq T1$ can always be passed where an object (an array of objects) of type $T1$ is expected.

In the second case, the actual parameter types are explicitly communicated to the parametric object in the form of class *meta-objects.* Also here, type safety cannot be ensured, since there can be a complete mismatch between the type represented by such a meta-object and the actual type parameter. Indeed, unlike in the implicit approach, compilation cannot even ensure that the meta-object represents a subtype of the actual type parameter.

A pragmatic approach. We have adopted an approach based on the latter of the two alternatives outlined above, however by automatizing the passing of the meta-object. The GJ compiler is "wrapped", that is, a first precompilation phase is added before invoking the GJ compiler itself. In that first phase, an array containing the class meta-objects for all actual type parameters in the order of their appearance is added as constructor argument. For example, a statement like

```
GDAC<T> ts = new GDAStrongSet<T>();
```

is transformed by adding an argument reflecting the meta-object(s) of the type parameter(s)

```
GDAC<T> ts = new GDAStrongSet<T>(new Class[]{T.class});
```

It is evident that this requires all constructors for generic classes to be defined with an additional argument, namely an array of class meta-objects. This also reflects its limitation: the signatures of type parameterized methods, also supported by GJ, would have to be modified, implying however more severe ramifications.

4.4 GDAC Classes

The previous section focused on the interface, through which an application can use our GDACs abstractions for publish/subscribe. As depicted earlier, our framework consists of a variety of GDACs spanning different semantics and guarantees, since different applications have different requirements.

4.4.1 Expressing Variations

These different semantics can be seen as QoS, and the various parameters are reflected by a variety of classes implementing the same interface. Hence, while im-

[5]The approach of passing an array is also mentioned in [BOSW98] as a *poor man's factory.*

plementations change, the interface remains the same. Figure 4.7 outlines different semantics implemented in the GDAC framework through different classes.

The framework is extensible, in that new classes can easily be added, (possibly extending existing ones) to implement new semantical choices. In particular, the classes presented in Figure 4.7 are abstract classes (though we use these classes as if they were concrete classes in the examples). Different concrete classes can extend such an abstract class, to provide the same semantics, yet with different implementations (e.g., using network-level or application-level protocols, see Chapter 6), also in different system models (e.g., *crash-stop* [FLP85] or *crash-recovery* [ACT00]). Due to these variable semantics, the notions of *order* and *reliability* used in the following are difficult to define rigorously for the entire framework.[6]

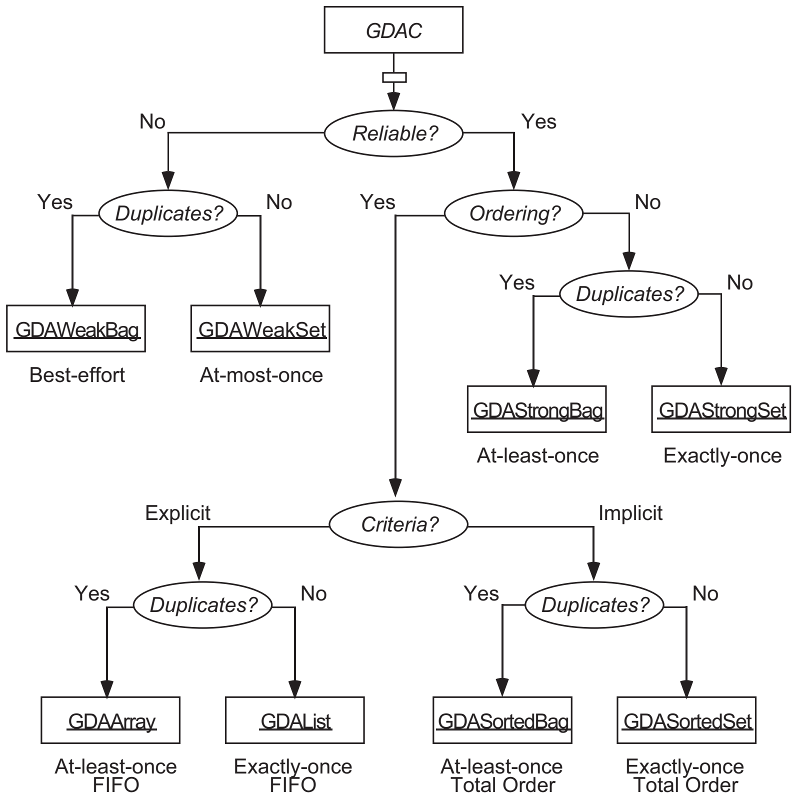


Figure 4.7: GDAC Framework

4.4.2 Order

Collections are often characterized by the way they store their elements. Plain sets or *bags* for instance do not rely on a deterministic order of their elements. Conversely,

[6]The decomposition of such protocols has been thoroughly studied in the context of group communication, for instance in *Horus* [vRBM96], or the *Bast* [GG00] and OGS (*Object Group Service*) [FG00] object systems developed in our lab.

sequences can store their elements in an order given explicitly, e.g., through indexes, or implicitly based on properties of the elements.

Storage vs Delivery Order

In GDACs however, the notion of space is somehow replaced by the notion of time. If some centralized collections reveal a deterministic storage order, a distributed asynchronous sequence may offer a deterministic ordering in terms of delivery to subscribers. In the Java collection framework for instance, a *sorted set* is a sequence which is characterized by an ordering of the elements based on their properties. This can be seen as an implicit order.

With our GDACs, an implicit order is a global delivery order on which the GDAC itself decides. The `GDASortedSet` class for instance presents a *total order* of delivery. Inversely, a *first-in-first-out* (FIFO) delivery order can be seen as an explicit order: it is given by the order in which obvents are notified to the GDAC by a publisher.

Insertion Order

In different centralized collections, the insertion order may have an impact on the storage order. In a *queue* or a *stack* for instance, the chronological insertion order will drive the storage order as well as the extraction order. A position can be given as additional argument to an insertion into a *list* for instance.

In an asynchronous collection however, the order of insertion can be interpreted as the order of sending (publishing). It seems however obvious that inserting an element at a specific position cannot be supported, since it would translate into delivering an obvent at a certain moment in time relative to other obvents; inserting an obvent at the beginning of a list would translate into sending that obvent before obvents that have possibly already been delivered to subscribers. Therefore there is never any explicit argument for the order passed when “inserting” a new element into a GDAC.

Extraction Order

Elements can be extracted from a centralized sequence, without any key nor explicit index, in mainly two ways, namely according to a FIFO order such as in queues, or in a *last-in-first-out* (LIFO) order corresponding to stacks.

Extracting obvents from a centralized collection translates into pulling obvents from a distributed asynchronous one. In the case of consumers polling a GDAC for new obvents, only a FIFO order ensures that the order determined by the underlying dissemination algorithm is respected when the obvents are effectively delivered to (pulled by) the consumer. One could however very well imagine to equip GDACs, which do not incorporate any notion of order of delivery, with a LIFO order with respect to pulling consumers. Nevertheless, we have chosen to adopt a FIFO order for all classes presented in Figure 4.7. A LIFO order can, if really required, still be implemented on top.

4.4.3 Reliability

For the sake of simplicity, we have always considered reliable delivery of obvents until now, i.e., a published obvent is received by all (correct) subscribers. Without loss of validity of the previously introduced models, we now distinguish more refined semantics. For instance, the `GDAWeakSet` does not offer more than plain unreliable delivery, whereas other classes guarantee reliability (e.g., `GDAStrongSet`). By distinguishing between unreliable and reliable GDACs, our framework hierarchy is roughly split into two subtrees, as depicted by Figure 4.7.

Duplicate Elements vs Duplicate Deliveries

While bags might typically contain duplicate elements, sets usually do not contain any duplicates. In the context of GDACs, duplicates can be considered in two places. First, similarly to centralized collections, when inserting obvents, and second, in delivering obvents.

According to the semantics of obvent publication defined in Section 3.3, the same obvent can be published twice, giving more freedom to the application. To respect this design choice, GDACs *must* accept duplicates. In the "traditional" sense, all GDACs hence represent bags. GDACs vary however by the way they handle duplicate *delivery*. Depending on the underlying algorithms, an obvent published once can give rise to several distinct clones of that very same obvent delivered to a given subscriber.

The simple `GDAWeakBag` class for instance does not prevent a notification to be delivered more than once, whereas the `GDAWeakSet` class gives stronger guarantees by eliminating duplicate elements. This property is orthogonal to other characteristics of our GDACs. For that reason, our framework contains a variant with and without duplicates for every other property, as shown in Figure 4.7. When allowing duplicates and combining with unreliable delivery for instance, the outcome is best-effort semantics. In return, with "reliable" delivery, *at-least-once* semantics can be guaranteed.

Persistence

Obvents may be volatile, which means that they may be dropped immediately after delivery. Conversely, the obvent could be stored in memory or even on persistent storage. In the context of this thesis, we did not deal with persistence of obvents. Obvents are considered volatile, and can even be dropped as soon as they have been delivered to the corresponding subscribers.[7] Missed obvents are therefore not

[7]Most implementations however buffer the last k delivered obvents that are still referenced by the application, k being configurable.

replayed to *late subscribers* or temporarily disconnected participants, though in the framework provided by the language integration approach presented in the next chapter, we have made room for this possibility at the programming level.

4.5 A Filter Library

We have used the model underlying content-based queries in TPS presented in Section 3.5 as a design pattern for devising a filter library. In short, we have implemented a set of conditions, which encapsulate different comparisons, and subscription patterns are built by customizing and combining such primitive filters. The resulting design pattern is schematically outlined in Figure 4.8.

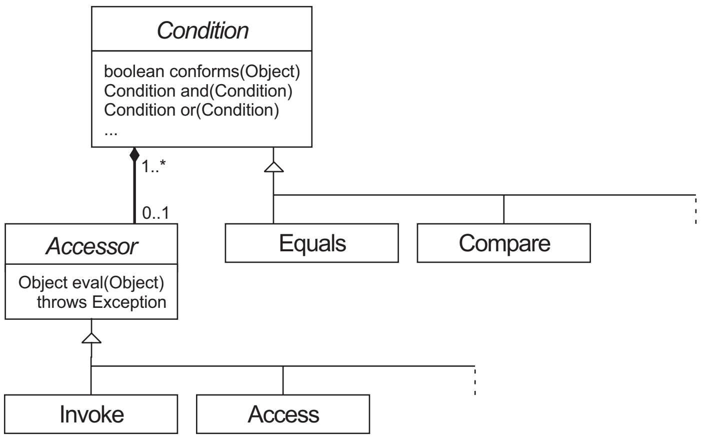


Figure 4.8: Types for Reifying Subscription Patterns

4.5.1 Accessors

Accessors are specific objects used to access partial information on the runtime event objects and are similar to the *function objects* defined in JGL.[8]

Querying Objects

In Java, an accessor object implements the interface `Accessor` given in Figure 4.9, and is evaluated by calling the `eval()` method with the event object as argument. This method can also throw exceptions raised when evaluating the method chain, which enables the reaction to exceptions. Returning `null` in case of exceptions would contradict the use of `null` as matching criterion.

[8]As we will show later, our accessors differ from function objects in JGL, in that they are used to represent parts of subscription patterns, and are hence devised in a way enabling optimizations.

```
package GDACs.Conditions;

public interface Accessor {
  public Object eval(Object e) throws Exception;
}
```

Figure 4.9: `Accessor` Interface

Reflection-Based Accessors

We have implemented accessors in Java using *structural reflection* [EG01a], one of the two main kinds of reflection pointed out in [Fer89]. While this structural reflection reifies the structural aspects of a program, such as data types, or obvents in our case, the second main kind of reflection, termed *behavioral reflection* (*computational reflection*), is concerned with the *reification* of computations and their behaviour.

Reflection in Java. The inherent Java language reflection capabilities [Sun00a] consist in a type-safe API that supports *introspection* about classes and objects in the current Java virtual machine at runtime. We view introspection as one aspect of structural reflection, limited to the reification, in the sense of *representation*, of structures of types and classes at runtime. A second aspect, the *modification* of those structures (also sometimes termed *intercession*) is not addressed by the Java core reflection API.[9]

Note that starting at Java 1.3, a limited mechanism for behavioral reflection has been added through the `java.lang.reflect.Proxy` class. We will come back to this in Section 4.7.3.

Implementing accessors. Java provides meta-objects which reify not only classes, but also methods, fields, constructors, etc. We make extensive use of meta-objects for methods (`java.lang.reflect.Method`) to reify the m_i's of accessors, as shown in the design pattern in Figure 4.8.

Such a meta-object representing a method defers to runtime the choice of *which* method is to be invoked, and enables also the actual performing of such a *dynamic invocation*.

In languages lacking meta-objects representing methods, specific objects can be generated either manually for the methods that will be used, or by a compiler.

[9]Javassist [Chi00] and OpenJava [TCKI00] are two approaches to instrumenting Java with *load-time* structural reflection, i.e., the ability of *modifying* classes at runtime prior to instantiation. OpenJava promotes a compiletime *meta-object protocol* (MOP) [Chi95] based on an extension of `java.lang.Class`, and makes use of the Sun Java compiler, while Javassist provides an extended class loader supporting the creation of new methods as copies of existing ones.

```
package GDACs.Conditions.Accessors;

import java.io.*;
import GDACS.Conditions.*;

public class Invoke implements Accessor, Serializable {
  /* specified by names */
  public Invoke(String methodNames, Object[][] ps) {...}
  /* specificied by methods */
  public Invoke(Method[] methods, Object[][] ps) {...}
  /* nested accessor, by name */
  public Invoke(Accessor nested, String methodName, Object[] ps) {...}
  /* nested accessor, by method */
  public Invoke(Accessor nested, Method method, Object[] ps) {...}
  ...
}
```

Figure 4.10: `Invoke` Class (Excerpt)

Implementation Examples

These meta-objects have enabled the implementation of the `Invoke` class shown in Figure 4.10. Accessors represented by instances of that class can be used with any obvent type. For illustration purposes, we have likewise also implemented an accessor enforcing direct attribute accesses, i.e., based on meta-objects representing fields (`java.lang.reflect.Field`). Figure 4.11 gives an outline of its API.

```
package GDACs.Conditions.Accessors.*;

import java.io.*;
import GDACs.Conditions.*;

public class Access implements Accessor, Serializable {
  /* specified by names */
  public Access(String fieldNames) {...}
  /* specificied by fields */
  public Access(Field[] fields) {...}
  /* nested accessor, by name */
  public Access(Accessor nested, String fieldName) {...}
  /* nested accessor, by field */
  public Access(Accessor nested, Field field) {...}
  ...
}
```

Figure 4.11: `Access` Class (Excerpt)

Specifying methods. Our accessor classes offer different constructors for convenience. The first two constructors for the two illustrated accessors enable the direct expression of a nested "access". The methods and attributes are represented either by name or by corresponding meta-objects. In the former case, names are separated by a "." as for instance in `"method1.method2. ... methodv"` illustrating nested methods.

We illustrate these two ways for an application to initialize accessors in the case of method invocations:

By method object: The application explicitly deals with reflection, and provides a `Method` object. As a direct consequence of Java's nominal subtyping, a method object m is bound to a single type T: if a method object m for type T is applied to an object o which does not conform to T, an exception is thrown, even if o implements a method of the same name and signature as m. By specifying methods as objects, the application implicitly defines the type of event objects it is interested in.

By method name (and signature): Specifying the name of a method and its actual arguments is much easier for the application developer. In TPS, the types of the expected obvents is known when subscribing to a GDAC, and the lookup of the corresponding method object (through `java.lang.Class`) by the accessor can take place automatically, once and for all. With Java reflection, the most suitable method for the actual arguments is obtained. Note however that during this lookup, only the *dynamic types* of the actual arguments are considered, i.e., the method chosen to effectively handle the invocation is the same than the one that is obtained in a static call with arguments of static types equal to the dynamic types of the actual arguments passed to the accessor.[10]

Mixing attributes and methods. The other constructors shown in Figure 4.10 and Figure 4.11 enable the creation of an accessor reflecting nested accesses by specifying an explicitly created nested accessor.

We have chosen this general design offering the possibility of breaking down an accessor into its single stages to enable an easy mixing of attribute accesses and method invocations inside the same condition.

Type Safety in Accessors

Knowing the type of the fitting event objects also represents a step towards type safety. If every m_i of an accessor is reified, the return type of each such m_i can be checked (at runtime) for its conformance to the type bound to m_{i+1}. Similarly, the type of each provided actual argument $p_{i,j}$ can be checked for its conformance to the type of the j-th formal argument of m_i. By enforcing these checks, the `Invoke` class rules out type errors.

[10]In languages offering *dynamic multi-dispatch* the obtained methods would be the same in both cases (see Section 5.5.2).

Avoiding type errors. Note that the implementation of these type checks relies strongly on the `isAssignableFrom()` method in class `java.lang.Class`. For that purpose, all arguments representing primitive types are transformed to their corresponding object counterparts. In general, when specifying accessors, a developer must always reason in terms of objects:

Method arguments: When specifying which methods are to be used by an accessor, actual values for formal arguments of primitive types have to be passed as their object counterparts. This can give rise to conflicts when specifying a method by its name and actual arguments. For example, during method lookup, an accessor has no means of knowing whether the developer was thinking of a method which takes a single parameter of type `int` or a single parameter of type `Integer`. A class which contains such a method pair gives rise to two method objects, with different meta-objects for their respective formal arguments. To correctly handle the case where only a method with a parameter of type `int` exists, the `Invoke` class is implemented in such a way that it tries one permutation of types/primitive types after another, until it finds a fitting method through introspection (if there is any).

Return type: Similarly, with reflection, a value of primitive type returned by a method call is transformed to its object variant, which is rather convenient. E.g., an invocation chain can contain a call to a method defined on the `Integer` type, while the previous method yields a value of `int`. In static code, an object of type `Integer` would have to be explicitly created. Note that, unlike with method arguments, this case could not lead to any ambiguities, since in Java, return types are not considered to be part of the method signatures (in the sense that one can not define two methods with corresponding formal arguments but different return types in the same class).

Achieving structural conformance with reflection. By specifying methods by their name and signature, a primitive form of structural conformance of types could be achieved by refraining from any type checks, hence deferring the method lookup to runtime. Prototype implementations of certain GDACs for that end implement a method `containsAll()` (borrowed from the original Java collections), with a signature similar to the `contains()` method for subscribing.

This however implies, for *each* evaluated event object, a costly *dynamic lookup* to obtain the appropriate method meta-object (see Section 4.5.4).

4.5.2 Conditions

While an obvent is queried through an accessor, a condition object evaluates the obtained information, i.e., decides whether the obtained object represents a desirable value for the corresponding subscriber.

In Java

In Java, a condition object implements the `Condition` interface given in Figure 4.5. It is evaluated for a given event object o by invoking `conforms()` with o as argument. The condition classes we propose are conceptually similar to the *predicates* found in JGL that are used in conjunction with centralized collections.

Comparisons

The different condition classes we provide vary by the comparison functions they encapsulate. The other attributes of conditions according to the model presented in Section 3.5.1, namely the used accessor and the expected result, can be viewed as initialization arguments of our condition classes.

In short, Java inherently implements the three basic comparison mechanisms outlined in the previous chapter. The comparison of two objects with the `==` operator (identity) yields `true` iff the two arguments are references to the same object, but is less useful in our context.

State (value). Every object can also be compared to any other object by means of the `equals()` method, which is inherent to all Java objects and can be overridden by application-defined classes. When overriding, one must be careful to guarantee a reflexive, transitive, consistent (cf. Section 3.5.1), and symmetric implementation. Supposing two objects o_1 and o_2, the latter property manifests itself in that calling the `equals()` on either o_1 or o_2 with the other object as argument yields the same result.

Ordering. This is for objects implementing the `java.lang.Comparable` interface, the counterpart to the well-known `Magnitude` type in Smalltalk [GR83]. Comparable objects in Java provide a method `compareTo()` which returns an integer, indicating the order of the object o_1 with respect to o_2 (usually `-1` if o_1 is smaller, `1` if bigger, `0` if equal). Such objects manifest a natural ordering, e.g., class `Integer`, and can thus be matched against a range of values. Comparisons can be moved out of the compared objects by using `java.util.Comparator` objects, which represent binary predicates.

Example Implementations

Hence, as outlined in Section 3.5.1, and as illustrated by the two most relevant comparisons in Java, b is represented by a method, and can also be viewed as m_v. Inversely, methods $m_u ... m_v$ $(1 \leq u \leq v)$ can be seen as part of the comparison.

`Equals`. In our condition classes, like the `Equals` class given in Figure 4.12 (representing a value-based equality test), we have added constructors which alleviate their use. The third constructor in the figure for instance enables the expression of nested method calls by providing a string denoting the names of the methods/attributes. The accessor is in that case created implicitly.

The increased flexibility offered by the possibility of mixing methods and attribute accesses however mandates that argument lists for methods be provided as (possibly empty) arrays. In contrast, the argument lists for field accesses are represented by `null` values when creating immediately a nested accessor. This is necessary for the condition to be able to distinguish between attribute accesses and methods, since an attribute and a method can have the same name in a given class.

```
package GDACs.Conditions;

import java.io.*

public class Equals implements Condition, Serializable {
  /* compare the event object as a whole */
  public Equals(Object to) {...}
  /* compare return value of accessor */
  public Equals(Accessor a, Object to) {...}
  /* implicit accessor creation */
  public Equals(String names, Object[][] ps, Object to) {...}
  ...
}
```

Figure 4.12: `Equals` Class (Excerpt)

`Compare`. Figure 4.13 outlines the `Compare` class, which implements a condition representing an order-based comparison of two objects. The first constructor expresses a comparison of the evaluated obvent itself with an object *to*. The result is compared to the integer value provided as argument. The comparison hence not only consists of an invocation of the `compare()` method, but also of a value-based comparison of the returned integer value with the provided one r.[11]

Similarly to the `Equals` class, the `Compare` class provides constructors which enable the implicit construction of the accessor. Furthermore, the `Compare` class provides constructors enabling the specification of an explicit comparator of type `Comparator`, as described above.

4.5.3 Subscription Patterns

In Java, a subscription pattern s is represented by an object of type `Condition`, and the function f indicating how to combine the outcome of the involved basic

[11]This scheme slightly diverges from the model presented in the previous chapter, since the `Compare` class is able of representing *any* simple comparison like $>$ or $=<$. For simpler use, our framework contains more specific classes, such as `Greater`, `LessEqual`, etc.

```
package GDACs.Conditions;

import java.io.*

public class Compare implements Condition, Serializable {
  /* compare the event object as a whole */
  public Compare(Object to, int r) {...}
  /* compare return value of accessor */
  public Compare(Accessor a, Object to, int r) {...}
  /* implicit accessor creation */
  public Compare(String names, Object[][] ps, Object to, int r) {...}
  /* compare the event object as a whole */
  public Compare(Object to, Comparator b, int r) {...}
  /* compare return value of accessor */
  public Compare(Accessor a, Object to, Comparator b, int r) {...}
  /* implicit accessor creation */
  public Compare(String names, Object[][] ps, Object to, Comparator b, int r)
    {...}
  ...
}
```

Figure 4.13: `Compare` Class (Excerpt)

conditions is not directly reified. In fact, f is explicitly constructed by *combining* conditions. These combinations are expressed through specific conditions, reflecting binary predicates, like `And` (Figure 4.14), `Or`, etc. Furthermore, we propose a unary predicate `Not` for inverting conditions.

Expressing Condition Combinations

For example, if the arguments `first` and `second` in Figure 4.14 passed to the constructor are not themselves composed conditions, then $w = 2$, and furthermore $f(b_1,\ b_2) = b_1 \wedge b_2$. To ease the expression of combinations, we have added methods for simple composition to the `Condition` interface. By calling the `and()` method, an instance of the `And` class in Figure 4.14 is implicitly created. This pattern counteracts Java's lack for operator overloading (see Section 4.7).

Type Safety

This subscription scheme based on conditions has the nice advantage of inherently expressing the subscription grammar. Syntax errors known from subscription languages, where they are only recognized at execution of the parser, are here detected by the Java compiler. Badly typed method and attribute names, on the other hand, are detected at runtime thanks to the provided type information in TPS.

```
package GDACs.Conditions;

import java.io.*;
import GDACs.*;

public class And implements Condition, Serializable {
  /* the two arguments */
  private Condition first;
  private Condition second;
  /* combine two subpatterns */
  public And(Condition first, Condition second)
    { this.first = first; this.second = second; }
  /* primitive evaluation */
  public boolean conforms(Object m)
    { return first.conforms(m) && second.conforms(m); }
  ...
}
```

Figure 4.14: `And` Class (Excerpt)

4.5.4 Performance Evaluation

Reflective systems and meta-level architectures offer increased modularity and flexibility. The benefit of such dynamism is often, but not necessarily, diminished by performance degradation.

Preliminary: Cost of Reflection

According to the way we have described our implementation, methods are invoked dynamically, i.e., through reified methods. Such dynamic invocations are much more expensive than static ones. Moreover, when subscribing to structurally conforming objects, method objects are obtained at runtime for each obvent. Such lookups are very costly, and are summed with the overhead of dynamic invocations.

Figure 4.15 depicts the cost of dynamic calls by comparing the overhead of local method invocations with a varying number of arguments (between 0 and 10 objects). These are performed using (1) dynamic invocations, each combined with a method lookup, (2) dynamic invocations without lookups, and (3) static invocations. These tests were made on a Sun Ultra 60 (Solaris 2.6, 256 Mb RAM, 9 Gb harddisk) with Java 1.2 (native threads). The test setting did not involve any *just in time* (JIT) compiler. The speedup factor observed for static invocations when using a JIT compiler was over three. The speedup in the case of dynamic evaluation is, as expected, insignificant.

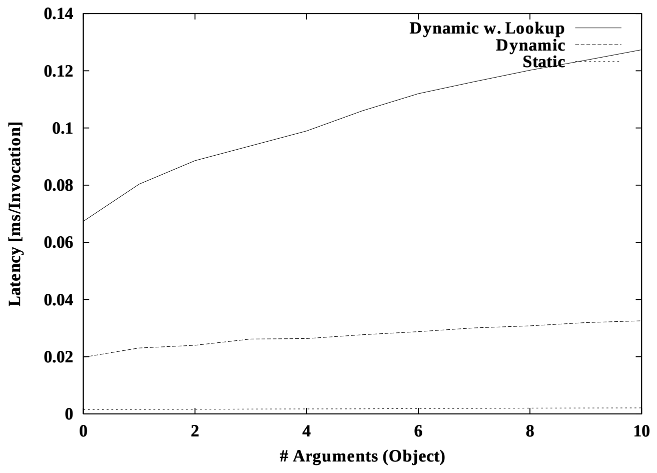


Figure 4.15: Latency with Different Invocation Styles

Optimizations

The type information available in TPS enables the application of optimizations when structural conformance is not required. We propose here two different optimizations.

Avoiding redundant invocations. Obvents are usually matched against patterns of several subscribers at a time, and these patterns are likely to present redundancies. We discuss here an optimization based on that observation, which is similar, but not identical to the *tree matching* algorithm used in Gryphon [ASS$^+$99]. The tree matching algorithm factors out redundant subpatterns with simplified assumptions: everything is decomposed into *and*s of basic conditions, and the latter ones are primitive comparisons of attribute values with predefined values.

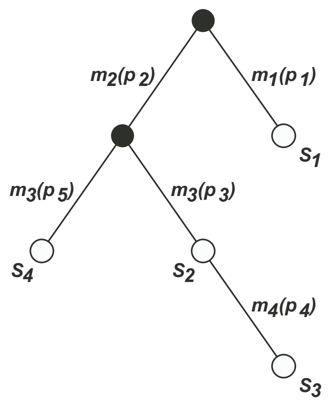
In contrast, our filter library offers more expressiveness, e.g., nested method invocations, different comparators and combinations (*and*, *or*, ...). Such combinations are performed statically, and dynamic queries on obvents represent the critical factor in our system. As a consequence, we focus on detecting common denominators of accessors, in order to avoid the evaluation of redundant dynamic method invocation chains.

Figure 4.16(a) shows a simple example of redundant accessors where each subscriber specifies a pattern consisting of a single basic condition. An *invocation tree*, like the one shown in Figure 4.16(b), is constructed from all accessors and is evaluated for every filtered event object.

Enforcing static filters. Based on the observation that dynamic invocations are far more costly than static ones, we have implemented an alternative optimization. Any dynamic invocations are avoided by generating static source code from accessors after performing type checks.

Subscriber S_1: $a_1 = (m_1, p_1)$
Subscriber S_2: $a_2 = (m_2, p_2), (m_3, p_3)$
Subscriber S_3: $a_3 = (m_2, p_2), (m_3, p_3), (m_4, p_4)$
Subscriber S_4: $a_4 = (m_2, p_2), (m_3, p_5)$

(a) Redundancy Between Accessors



(b) Resulting Invocation Tree

Figure 4.16: Optimizing Accessors

The generated source code is then directly compiled, by calling the original Sun Java compiler (`sun.tools.javac`), in a way similar as this is done in [TCKI00] or [KMS98].[12]

Obviously, this technique could be combined with the first optimization. We have however not pursued this approach here. The resulting static compound filter would have to be recompiled every time a single subscription pattern is modified.

Evaluation

We evaluate here the benefits of the two above optimizations by comparing the resulting performances with two non-optimized scenarios. These are namely (1) the filtering of structurally conforming obvents, and (2) the filtering of obvents conforming by name.

Testbed. Our measurements were made with the Java virtual machine 1.2, enabled JIT and native threads on Sun Solaris 2.6. A single producer was publishing obvents encapsulating a single string from one network (Sun Ultra 60, 256 Mb RAM, 9 Gb harddisk), to subscribers equally distributed over two further networks; one composed of altogether 60 Sun SuperSparc 20 stations (model 502: 2 CPU, 64 Mb RAM, 1Gb harddisk), and the second one composed of 60 Sun Ultra 10 (256 Mb RAM, 9 Gb harddisk) stations. The individual stations and the different networks were communicating via 100 Mbit Ethernet.

[12][KMS98] terms this technique (runtime) *linguistic reflection*, which is seen as a synonym of structural reflection.

Parameters. We have made a set of extensive tests, in which we have always varied one of four parameters for the subscriptions. These are namely, (1) the average fraction of positive matches for any subscriber p_1, (2) the total number of subscribers n, (3) the maximum nesting level of invocations for queries v, and (4) the number of different query methods w at each nesting level.

Varying p_1: From 100 published obvents, an average of $100p_1$ obvents matched a given subscribers pattern. Figure 4.17 shows the effect of varying p_1. It confirms the intuition that the cost of sending obvents with UDP does not depend on the matching scheme, and can be seen as fixed. With $p_1 < 0.01$ in this scenario, the pure cost of matching is measured. In order to accentuate the differences between the matching schemes without contradicting our concrete applications, we have chosen $p_1 = 0.1$ for the following measurements.

Varying n: Similarly to the scenario in Figure 4.16, we have chosen one basic condition per subscriber. Figure 4.18 reports the effect of scaling up n, conveying that the two optimizations are almost equivalent with a large n.[13] As shown in the previous figure, UDP is a limiting factor with an increasing number of sends (here due to a large n). Performance drops faster with static filters, since every additional subscriber involves a full pattern evaluation. In contrast, the optimized dynamic scheme is less sensitive since redundant queries are avoided.

Varying v: The probability of having i nested invocations was chosen as $1/(v+1)\ \forall i \in [0..v]$. Increasing v reduces throughput with static invocations, as illustrated in Figure 4.19, since static accessors comprise more invocations. Similarly, the optimized dynamic scheme is less efficient with an increasing v, since the total number of performed methods increases with the depth of the tree.

Varying w: One of w methods was chosen at each nesting level with a probability of $1/w$. Varying w obviously does not influence static filter evaluation. On the other hand, increasing w might lead to increasing the potential number of edges leaving from any node in the invocation tree. The resulting performance loss is directly visible in Figure 4.20. The optimized dynamic scheme is however more penalized by increasing w, as shown in the previous figure. This is due to the fact that increasing v by 1 might result in up to d new edges in every former leaf of the invocation tree.

Interestingly the optimized dynamic matching scheme never outperformed the static scheme, even if the speedups became close with a large number of obvents published. One could believe that with a strong redundancy between patterns, and a large number of subscribers, the dynamic scheme would become more efficient. Even with extreme parameter values, we have however never encountered such a scenario.

[13]As we will elucidate in Chapter 6, a possible dissemination algorithm relies on a hierarchical topology of participants, among which membership information is split up. A single participant rarely considers more than 100 participants when matching an obvent.

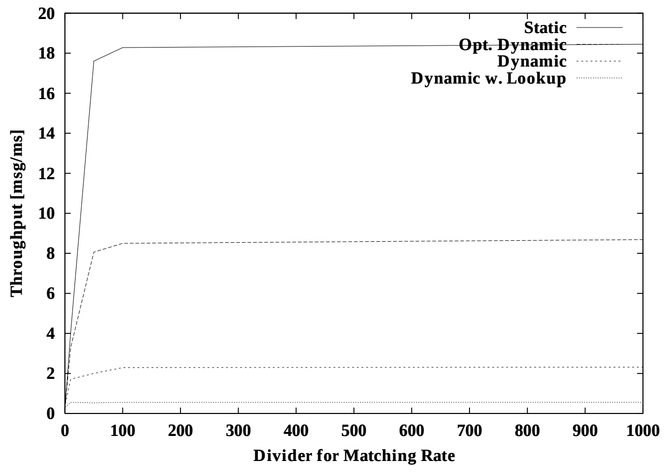


Figure 4.17: Varying the Matching Rate p_1 ($v = 0$, $n = 20$, $w = 1$)

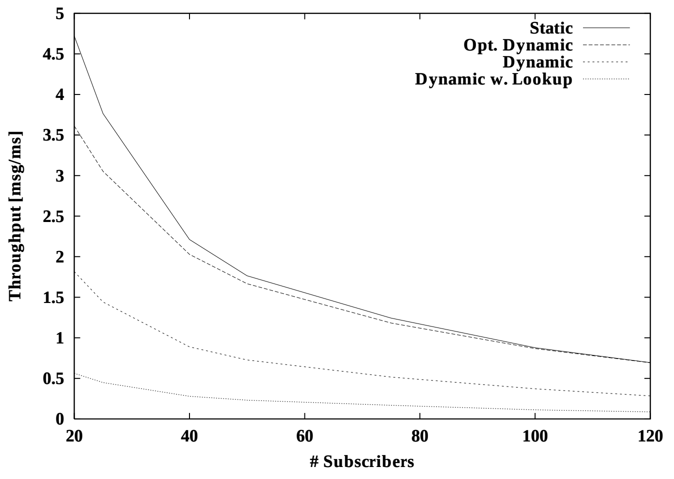


Figure 4.18: Varying the System Size n ($p_1 = 0.1$, $w = 1$, $v = 2$)

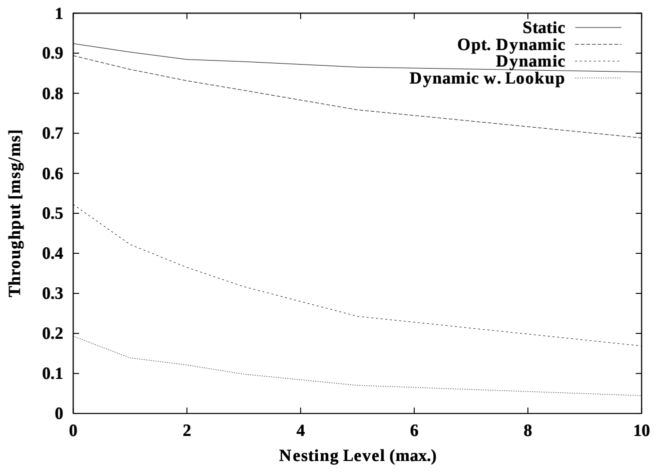


Figure 4.19: Varying the Complexity of Subscriptions v ($p_1 = 0.1$, $n = 100$, $w = 1$)

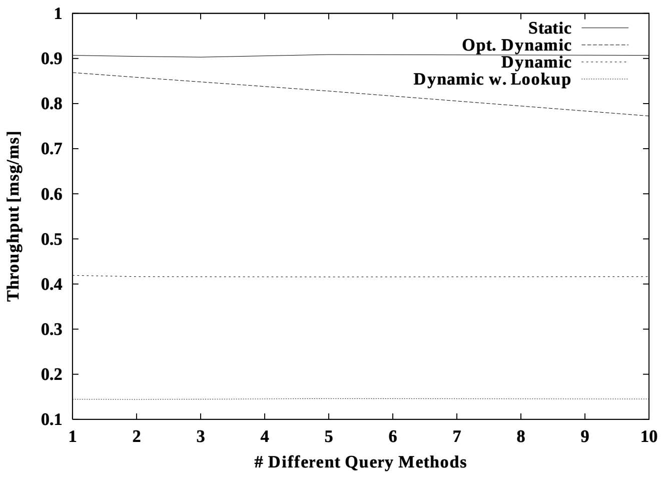


Figure 4.20: Varying the Redundancy of Subscriptions w ($p_1 = 0.1$, $n = 100$, $v = 1$)

4.6 Programming with GDACs: The Stock Trade Example

We illustrate the convenience of distributed programming with GDACs through a simple stock trade example. We first introduce the stock trade example in the face of TPS, and then focus on how to program the example with GDACs.

4.6.1 Preliminary: Stock Trade in TPS

As depicted in Figure 4.21, the communication medium is pictured as consisting of nested channels for obvent types.

Stock Obvents

Obvents are propagated on the corresponding channels, and subscriptions which are expressed for types are issued to the appropriate channels.

The example encompasses several obvent types. According to the subtyping rules in Java, by subscribing to a type `StockObvent`, one receives instances of its subtypes `StockQuote` and `StockRequest`, and hence all objects of type `SpotPrice` and `MarketPrice`.

Scenario

A possible scenario is the following. The stock market p_1 publishes stock quotes, and receives purchase requests. These can be "spot price" requests, which have to be satisfied immediately, or "market price" requests for purchasing quotes only at the end of the day, or once another given criterion is fulfilled. As outlined in Figure 4.21, these different kinds of events result in corresponding obvent types, rooted at the `StockObvent` type (details of the elaborate obvent types are omitted here for simplicity).

Market price requests can however expire, and for the broker's (e.g., p_2) convenience, an intermediate party (p_3), e.g., a bank, might also handle such requests on behalf of her/him, for instance by issuing spot price requests to the stock market once the broker's criteria are satisfied. Figure 4.21 illustrates this through p_2, which expresses only interest in stock quotes that cost less than \$100.

4.6.2 Stock Trade with GDACs

We illustrate how to put the stock trade application to work with GDACs. Figure 4.22 depicts two simple stock obvent types used in this example, corresponding to the types roughly outlined in Figure 4.21.

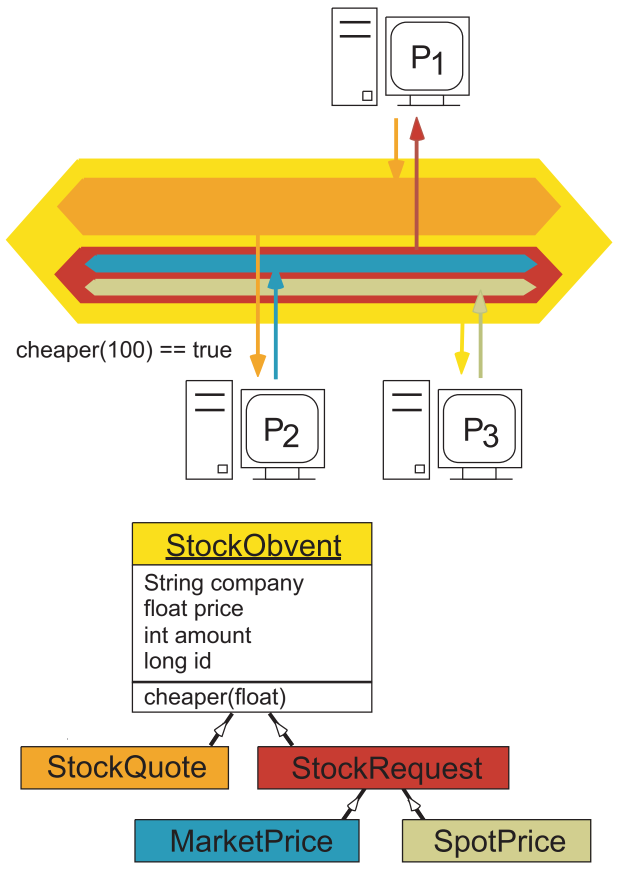


Figure 4.21: The Stock Trade Example in TPS

Publishing Stock Quotes

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

```
GDAC<StockQuote> quotes = new GDAStrongSet<StockQuotes>();
StockQuote q = new StockQuote("Telco Mobiles", 80, 100, 12373);
quotes.add(q);
```

Subscribing to Stock Quotes

Expressing a subscription requires more effort. A notifiable type has to be first created to handle callbacks:

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

```
import java.io.*;

public abstract class StockObvent implements Serializable {
  private String company;
  private float price;
  private int amount;
  private long id;
  public String getCompany() { return company; }
  public float getPrice() { return price; }
  public int getAmount() { return amount; }
  public long getId() { return id; }
  public boolean cheaper(float than) { return price < than; }
  public StockOvent(String company, float price, int amount, long id)
  {
    this.company = company;
    this.price = price;
    this.amount = amount;
    this.id = id;
  }
}

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

Figure 4.22: Simple Stock Obvent Types

Subscription patterns are expressed by combining conditions from our filter library. Here we express interest in stock quotes with a price less than $100 for the entire *Telco* group, not only the mobile division:

```
GDAC<StockQuote> quotes = new GDAStrongSet<StockQuote>();
Object[][] args1 = new Object[][]{{null},{"Telco"}};
Condition notContained =
  new Equals(".getCompany.indexOf", args1, new Integer(−1));
Object[][] args2 = new Object[][]{{new Float(100.0)}};
Condition cheaper = new Equals(".cheaper", args2, new Boolean(true));
Condition pattern = notContained.not().and(cheaper);
quotes.contains(new StockQuoteSubscriber(), pattern);
```

Note that the second condition expressed through the `cheaper()` method implemented by all stock quote obvents could be similarly expressed by querying the price of stock quotes and comparing the obtained value through an instance of the `Compare` condition.

```
...
Object[][] args2 = new Object[][]{{}};
Condition cheaper = new Compare(".getValue", args2, new Float(100.0), −1);
...
```

4.6.3 RMI and TPS: Hand in Hand

TPS is a distributed programming paradigm which perfectly fits modern application's need for mass dissemination of information. Again, any implementation of TPS in Java is not to be viewed as conflicting with Java RMI, but rather complementary. We illustrate this by making the two interaction paradigms collaborate to complete the above stock trade example.

Improving Stock Trade

Though the use of a publish/subscribe interaction for the dissemination of stock quotes seems appropriate by scaling easily to many brokers, it might seem more appropriate in certain cases to use a synchronous interaction with the stock market when purchasing stock options, e.g., a remote method invocation to ensure that the order has been received.

Figure 4.23 illustrates an extended `StockQuote` type, and a subscriber type. When publishing stock quotes, the stock market simply adds a remote reference to an object capable of handling purchase requests. When receiving stock quotes, the broker can thereby directly interact with the stock market to initiate a purchase.

Limitations

To correctly handle the serialization of remote objects (in the sense of Java RMI), the Java RMI serialization mechanism is used, enabling a transparent integration of RMI and TPS.

However, the current implementation of Java RMI presents a severe caveat, which becomes especially visible through this integration with obvents. In fact, the distributed garbage collection keeps a remotely accessible object from being garbage collected as long as there is at least one proxy for that object. When publishing an obvent containing a reference to a remote object, such a proxy is created for each subscriber, which can sum up to several 1000's. Every time a proxy is garbage collected, the Java virtual machine hosting the represented object is notified. Consequently, if a single subscriber crashes, the remote object will never be garbage collected. With a "weaker" implementation of Java RMI, such as the one proposed in [CNH99], this problem could be circumvented.

4.7 Discussion

We discuss here the expression of alternative communication styles with our collections, and also possible enhancements.

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

/* stock broker */
public interface StockBroker extends Remote {...}

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

/* full stock quotes */
public class StockQuote extends StockObvent {
  private StockMarket market;
  public Stockmarket getMarket() { return market; }
  public StockQuote(String company, float price,
                    int amount, long id, StockMarket market)
  {
    super(company, price, amount, id);
    this.market = market;
  }
  ...
}

/* a stock subscriber */
public class StockQuoteSubscriber implements Subscriber<StockQuote> {
  private final StockBroker broker = ...; /* reference to this broker */
  public void notify(StockQuote q) {
    System.out.print("Got offer: ");
    System.out.println(q.getPrice());
    ...
    boolean bought =
      q.getMarket().buy(q.getCompany(), q.getPrice(),
                        q.getAmount(), broker);
    ...
  }
}
```

Figure 4.23: Advanced Stock Obvent Types and Subscriber

4.7.1 Subject-Based Publish/Subscribe

As already explained in the previous chapter, one can quite obviously devise a subject-based scheme on top of TPS, since TPS includes a form of content-based subscription, e.g., by designing an obvent type (possibly the root obvent type) which contains an attribute reflecting a subject name.

Retrospective: DACs

We would like to recall that the GDACs presented in this chapter are descendants of the (non-generic) *Distributed Asynchronous Collections* (DACs) originally presented in [EGS01]. Indeed, these were initially intended as general abstractions for decoupled communication between distributed producers and consumers, and [EGS01] presented an illustration through subject-based publish/subscribe.

Expressing Subjects with DACs

A DAC basically represents a channel for unbound objects of a certain kind, and this event kind can also be easily mapped to something different than an event type, like a subject.

Subject Scheme. This can be implemented by adding a subject name as attribute to the DAC itself, and initializing it upon creation of the DAC. Typically, a nested subject name such as "/Chat/Insomnia" is a reference to the subject called "Insomnia" which is a nested subject of "Chat", and all events published through a DAC created for "Chat/Insomnia" implicitly belong to that subject "Chat".

The root of the hierarchy is represented by an *abstract subject* (denoted by "/"). Top-level subjects, which are no specializations of already existing ones, are contained in the abstract subject only. Subscribing to a subject triggers subscriptions to the contained subjects as well.

Subscribers. With the push model adopted in DACs, subscribing entities must register a callback object. That callback object must implement a specific interface, namely the (non-generic) `Subscriber` interface, shown in Figure 4.24. Through a call to the `notify()` method, the DAC notifies the subscriber that it contains a new notification. Note that the second argument is required since an event does not implicitly carry the subject it was published to. As a consequence the same callback object can be used here for several unrelated subjects, while with GDACs this is not possible because of the type parameter (and its implementation in GJ). With two distinct types *T1* and *T2*, even if *T2* $\leq$*T1*, a subscriber for type *T2* is not a subscriber for *T1* ($\forall R, T_1, T_2 : R\langle T_1\rangle \;\leq\; R\langle T_2\rangle \Leftrightarrow T_1 = T_2$). This increased flexibility is however neglectable regarding the loss in type safety.

```
package DACs;

public interface Subscriber {
  public void notify(Object o, String subjectName);
}
```

Figure 4.24: `Subscriber` Interface used with DACs

Combining Subjects and Types

Though not presented here, one could easily imagine a mixture of subjects and types, by providing GDACs for specific subjects. This could lead to several separate spaces for an obvent type T; one for each subject for which there exist GDACs parameterized by that same obvent type T.

4.7.2 Message Queuing

As outlined in Section 2.3.2, the shared space paradigm has given rise to more industrial strength variants, commonly termed "message queues".

Removing Future Elements. To express such queuing interaction with GDACs, we overload the `remove()` primitive known from conventional Java collections. This reflects an asynchronous removal of objects, where objects can be "removed" though not yet present: future elements added to the GDAC are delivered and successively removed from the GDAC.

Callback. As outlined through GDACs, the all-of-n semantics, characteristic for "traditional" publish/subscribe can also be provided as pull-style interaction of the consumers with the channel. Inversely, message queuing (one-of-n) can be implemented with callbacks to consumers. In other terms, the choice of push- or pull-style interaction of consumers with a shared space is orthogonal to the dissemination model.[14]

GDACs consequently implement several variants of the `remove()` method, as shown in Figure 4.25.

[14]Note that we have abandoned the model introduced in [EGS01], which artificially separated message queuing (one-of-n) and publish/subscribe (all-of-n) in two distinct subtype hierarchies. As successfully shown by the tuple space, one-of-n and all-of-n can be merged inside the same abstraction, by keeping certain possible conflicts in mind.

```
package GDACs;

import java.util.*;

public interface GDAC<T> extends Collection<T> {
  ...
  /* one-of-n, pull-style */
  public void remove(ExceptionHandler h, Condition c);
  public T remove(ExceptionHandler h);
  /* one-of-n, push-style */
  public void remove(Subscriber<T> t, Condition c);
  ...
}
```

Figure 4.25: `GDAC` Interface (Second Excerpt)

4.7.3 Ensuring Type Safety of Patterns at Compilation

The outlined filter library implemented with structural reflection faithfully reflects the model of subscriptions outlined in the previous chapter, and furthermore represents a very flexible solution, in that methods can be specified as meta-objects, or simply by their name (and signature). In the latter case, the system can perform the lookup of the corresponding method immediately, inherently performing type checks.

Static Type Checks

In the “normal” case, methods are specified according to the second possibility, i.e., by name and actual arguments, and type checks are performed, however only at runtime with introspection. In contrast, the parametric polymorphism applied to GDACs ensures type-safe obvent delivery at compilation. Hence it would make much sense to perform type checks on filters at compilation, e.g., through the Java compiler itself.

Exploiting Behavioral Reflection in Java

A starting point for static type checks on subscription patterns is given by a limited mechanism for behavioral reflection which made its way into Java version 1.3. In short, while meta-objects representing methods enable to decide at runtime only *which invocation* to perform, the `java.lang.reflect.Proxy` class enables to defer to runtime the decision of *what effect* an invocation should have. An instance of this proxy class can thus be used at runtime as a meta-object, for instance to perform computation prior to, and following, the invocation of the actual target object.

Expressing Conditions with Proxies. In our case, such proxy objects can be used to “register” the invocations used later to query objects. The application depicts what methods it wants obvents of type T to be queried through by (statically) invoking them on proxy objects provided by a GDAC for type T. The proxy simply registers these invocations and the corresponding arguments at runtime, and generates the subscription pattern. This scheme could be implemented by having the asynchronous `contains()` method for subscription return a proxy for type T, as outlined in Figure 4.26.

More precisely, the instance of T returned by the `contains()` method could then be used to apply an invocation, which would be registered by the handler. The return value of the handler would again be a new handler (for nested invocations), until a boolean value would be found. Consider the following illustration with stock quotes:

```
GDAC<StockQuote> g = new GDAStrongSet<StockQuote>();
StockQuote proxy = g.contains(new StockQuoteSubscriber());
proxy.getCompany().equals("Telco Mobiles");
```

Here, the `proxy` object holds an instance of the `Proxy` class defined as part of the Java core reflection API. A condition is simply expressed through `proxy`, by invoking the right methods in a nested way, yet finishing by a method returning a boolean value.

Note that the method selection is made statically by the compiler, in contrast to the dynamic method selection explicitly performed in the case of the filter library, i.e., based on the dynamic types of invocation arguments.

Limitations. The above illustration of the use of behavioral reflection through proxies would however not work, for the simple reason that Java currently only supports proxies for interfaces, i.e., only for abstract types, and not for classes. As a consequence, all return types of methods of obvent types would currently have to be interfaces, and GDACs could only be parameterized by interfaces as well, meaning that for virtually any concrete obvent type there would have to be an explicit interface defined, and subscriptions to classes would not be possible. As a consequence, built-in classes like those representing primitive object types could not be used, and we would have to introduce an alternative type hierarchy for the use with GDACs, rooted by an abstract type like the `ObjectIntf` type given in Figure 4.27.

Introducing own primitive object types could however in addition solve the problem of expressing subscription patterns as combinations of simple conditions, by integrating binary functions, such as `and()`, or `or()` into the new boolean type:

```
GDAC<StockQuote> g = new GDAStrongSet<StockQuote>()
StockQuote proxy = g.contains(new StockQuoteSubscriber());
BooleanIntf first = proxy.getCompany().equals("Telco Mobiles")
BooleanIntf second =
 proxy.getValue().compareTo(new FloatImpl(100.0)).equals(new IntegerImpl(1));
first.and(second);
```

The obvious flaw with this approach is that anyone could provide its own implemen-

```
package GDACs;

import java.lang.reflect.*;

public class GDAStrongSet<T> implements GDAC<T> {
  private Class TClass;

  public GDAStrongSet(Class[] classes) { this.TClass = classes[0];}

  public T contains(Subscriber<T> ts) {
    return (T)Proxy.newProxyInstance(TClass.getClassLoader(),
                                     new Class[] { TClass },
                                     new Registrar(new Context(), 1));
  }

  private class Context {...}

  private class Registrar implements InvocationHandler {
    private Context context;
    private int level;

    public Registrar(Context context, int level) {
      this.context = context;
      this.level = level;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable
    {
      /* register the invocation */
      ...
      /* return a new handler */
      Class returnClass = method.getReturnType();
      if (returnClass.equals(Class.forName("java.lang.Boolean")) ||
          returnClass.equals(Boolean.TYPE))
            return new Boolean(true);
      ...
      return Proxy.newProxyInstance(returnClass.getClassLoader(),
                                    new Class[] { returnClass },
                                    new Registrar(context, level + 1));
    }
  }
  ...
}
```

Figure 4.26: Statically Type-Safe Filters with Behavioral Reflection

tation of these abstract types. This is probably the main underlying rationale to supporting proxies only for interfaces. A proxy declaration for a final class (which cannot be inherited from) would yield a subclass of that class, jeopardizing the security aspects often underlying the definition of final classes.

```
package GDACs.PrimitiveObjects;

/* interfaces */
public interface ObjectIntf {
  public BooleanIntf equals(ObjectIntf o);
  ...
}
public interface BooleanIntf extends ObjectIntf {
  public BooleanIntf and(BooleanIntf b);
  public BooleanIntf or(BooleanInt b);
  ...
}
public interface NumberIntf extends ObjectIntf {...}
public interface IntegerIntf extends NumberIntf {...}
public interface FloatIntf extends NumberIntf {...}
...

/* classes */
public abstract class ObjectImpl implements ObjectIntf {...}
public class BooleanImpl implements BooleanIntf {...}
public abstract class NumberImpl implements NumberIntf {...}
public class IntegerImpl implements IntegerIntf {...}
public class FloatImpl implements FloatIntf {...}
...
```

Figure 4.27: Defining Own Primitive Object Types

Improvements

We summarize the current limitations which make the viability of the above scheme as a solution for transparent subscription pattern description rather questionable. Improvements could address the following:

Class proxies: Restricting developers to use only interfaces in obvent types, and furthermore only subscribing to abstract obvent types appears to be too restrictive. A more powerful model could be imagined, by having Java support proxies for all non-final classes, possibly even including a control over proxy creation for final classes in Java's security framework.

Operators: Operators for primitive types should be reflected by methods in the corresponding object types, enabling proxies to intercept these calls. This can for instance be implemented as part of operator overloading, like in C++. That way, if an invocation chain returns an integer object that object can be compared with another integer object, also after having been increased.

Primitive types: Another consequence of this missing operator overloading in Java is that primitive types cannot be fully supported as return types of methods. This would require that operators applied to such primitive types can be overloaded as outlined above. Java proxies are very well able of supplying an instance of a primitive object type instead of a value of the primitive type, but the method implementing an operator would have to be furthermore invoked on the corresponding primitive object class to make it interceptable.

The last two are to some extent redundant. Indeed, if all operators for primitive types are implemented as methods on the corresponding primitive object types, there is no need for primitive types as return types. Nevertheless, it would be very convenient if operators applied to primitive types would be automatically transformed to the corresponding method calls on the primitive object type, and hence on a proxy.

Whether operator overloading should or will be added to Java is another question.[15] From our point of view, it would obviously make subscription patterns more intuitive. However, if the language does not support this, then this lack appears in various situations, not only in subscription patterns, and hence Java programmers are prepared for this.

[15]In 1998, operator overloading in Java represented an active research area. The Numerics Working Group at the Java Grande Forum invested quite some effort in this issue, and Bill Gosling himself seemed to encourage an adequate extension of Java. It seems however that this possibility has met a serious reluctance, and the issue has been silently dropped.

Summary

GDACs represent a library approach to expressing TPS in Java. Similarly to established abstractions for subject-based publish/subscribe, GDACs represent first-class channels. However, GDACs come as a framework integrated with the standard Java collection framework, where the different GDAC (sub)types are used to express different QoS.

The implementation of GDACs has benefited strongly from the default serialization mechanism offered as part of Java, as well as from structural reflection. The latter mechanism has allowed subscription patterns to respect the model presented in Chapter 3, by querying obvents through methods. The subscription grammar is expressed through an API, hence making use of the Java compiler to detect many possible errors. Nevertheless, type safety is only ensured at compilation, since methods used for querying are defined dynamically. Also, subscription patterns expressed through our filter library, though reflecting Java code, have only little in common with the native Java syntax.

Chapter 5

Java$_{PS}$: Integrating TPS into the Programming Language

In this chapter, we describe a second approach to putting TPS to work in Java. This solution consists in augmenting the Java language with two primitives, `publish` and `subscribe`, in order to inherently support TPS. We refer to this dialect of Java as Java$_{PS}$ [EGD01].

Our language integration approach benefits strongly from Java's built-in serialization mechanism, and the multiple subtyping provided through interfaces. The main additions made to the Java language, for the sake of supporting TPS, are *closures*, together with a very specific form of *deferred code evaluation.*

In order to avoid making the Java virtual machine distribution-aware, invocations of our primitives are transformed at compilation to calls to specifically generated classes encapsulating the effective communication infrastructure, implemented as part of our DACE project. These adapter classes can be viewed as a form of proxies for TPS. Inherent types and classes are regrouped in a package which we denote by `java.tps`.

5.1 Overview

We overview the driving forces behind the idea of integrating primitives into the Java language, and give an informal overview of these primitives.

5.1.1 Motivations: Lessons Learned from the Library Approach

The library approach is interesting from the point of view that it helps illustrate that TPS can be implemented in a type-safe way with an "ordinary" programming language such as Java. Nevertheless, our library approach turns out to present mainly two flaws, besides the fact that subscription patterns expressed through the filter library are not statically type-safe.

QoS

With GDACs, the only contracts between remote parties are the obvent types. By publishing obvents of a type that matches a subscriber's criteria, a publisher is implicitly (though weakly) linked with that subscriber. The QoS related to the interaction are defined on both sides through the types of the corresponding GDACs, yet are not agreed upon. Indeed, suppose a producer publishing obvents of a type T through a GDAC which is of a concrete type `GDABag`, ensuring only best-effort handling of obvents. A consumer might very well subscribe to T through a GDAC of type `GDASet`, expressing interest in reliable delivery of all instances of T. How to reconcile these two diverging views?

There are certainly ways of negotiating between different QoS requirements. The resulting QoS however represent a compromise, which might strongly differ from individual demands. In particular, it is difficult to define a general strategy to find a common ground between different requirements, especially in a more realistic scenario involving more than one publisher and more than a single subscriber for a same type of obvents.

Channel Management

Part of this problem arises because the channel management is made explicitly in the library approach, e.g., GDACs are explicitly created. This is in general an unnecessary burden. A producer wanting to publish obvents should not have to create a proxy before doing this, but should be able to simply "fire" obvents, and inversely, a subscriber should not be forced to subscribe to a channel representing the desired type. Ideally, a subscription should be expressible through the type itself.

5.1.2 Java Primitives for TPS

These deficiencies are addressed by the current language integration approach. We overview the two primitives added to the Java language.

Publishing

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

```
publish o;
```

This statement triggers the creation of a copy of o for every subscriber, according to the rules described in Section 3.3. In that sense, the action of publishing manifests its strong relationship with the `new` primitive found in many languages, by appearing as a similar primitive. Syntactically however, the `publish` primitive bears more resemblances with the very common `return` primitive, or the `throw` primitive for the raising of an exception in Java, by taking an object as argument. The main difference consists in that the intended receiver is not the invoker (e.g., of the encapsulating method body), but a wider audience, given by the set of subscribers with matching criteria.

Note here that exceptions in Java are subtypes of the root object type, just like our obvents.

Subscribing

A subscription encompasses a type T, as well as the desired properties of obvents of that type T. Such a subscription pattern appears directly as a predicate, or filter. We thus combine a subscription to a type T with the declaration of a filter, and an obvent handler:

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

The first expression delimited by brackets represents a block, provided by the application, which expresses which obvents of type T (represented by a formal argument called t here) are of interest. The evaluation of this first block yields a boolean value. The second expression is a block which is evaluated every time an obvent successively passes the filtering phase, and corresponds thus to the subscriber object. The same formal argument t represents the obvent of interest in this case.

The `subscribe` primitive only enables the expression of a subscription. Its activation, deactivation, as well as the setting of certain parameters, requires a subscription handle, which is returned by the `subscribe` primitive.

5.2 Publishing

An obvent can be published, which means that it will be asynchronously sent to any concerned subscriber. Following the Java language specification [GJSB00] based on a LALR(1) grammar, we introduce below a new statement for publishing obvents.

5.2.1 Syntax

A *PublishStatement* conforms to the following syntax. Figure 5.1 shows in more details how it fits into the Java language syntax (definitions added are typed in **bold** font).

PublishStatement:
  `publish` *Expression* ;

Here *Expression* is a non-`null` expression of a specific type `Obvent`, as opposed to the library approach. Though most class libraries relying on Java serialization use formal parameters of the root type `java.lang.Object`, yet expect an object of a specific type `java.io.Serializable` (throwing an exception at runtime if the actual argument is not serializable, Section 3.3.3), core language features strive for static type safety. For instance, all exceptions in Java are subtypes of the `Throwable` type, which subtypes the very root object type. Only instances of such specific types can be thrown, and this is verified at compilation both in method headers and bodies.

Nevertheless, an exception of type `CannotPublishException` can be thrown by the `publish` primitive at runtime, signalling any problems in transmitting the obvent. Figure 5.2 summarizes the basic exceptions.

Statement: § 14.5
  StatementWithoutTrailingSubstatement
  ...

StatementWithoutTrailingSubstatement:
  Block
  EmptyStatement
  ExpressionStatement
  SwitchStatement
  DoStatement
  BreakStatement
  ContinueStatement
  ReturnStatement
  SynchronizedStatement
  ThrowStatement
  TryStatement
  PublishStatement

PublishStatement:
  `publish` *Expression* ;

Figure 5.1: Syntax of Publish Statements in Java$_{PS}$

```
package java.tps;

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

Figure 5.2: Exceptions in Java$_{PS}$

5.2.2 Obvents

Remember that a key concept of TPS is to view events as application objects, and as illustrated through the previous chapter, this can be achieved fairly easily in

Java by exploiting its built-in serialization mechanism. Accordingly, obvents are serializable Java objects, and unlike in the library approach, a root `Obvent` type, subtyping `Serializable`, is explicitly introduced here. The motivations for this are given by (1) the desire to illustrate that obvents are *specific* unbound objects (and to statically type-check these as explained above), and (2) issues concerning their implementation (see Section 5.4.2).

```
package java.tps;

import java.io.*;

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

Figure 5.3: Obvents in Java$_{PS}$

QoS and Obvents

Obvents can serve different purposes. Distributed applications have diverging requirements: due to the inherent difficulty of combining (1) strong guarantees and (2) high performance in a large scale asynchronous distributed system such as the Internet, high throughput is often combined with “primitive” guarantees, while strong guarantees are usually more expensive to implement. Beyond this tradeoff and the feasible implementations, we foresee different characteristics for obvents. [1]

Delivery semantics. The first kind of characteristics are the *delivery semantics* associated with obvents; an expression of quality of delivery.

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

[1]Note that the QoS framework elucidated in the following differs from the one proposed in the library approach.

Reliable: Once successfully published, a reliable obvent will be received by any subscriber that is "up for long enough". A subscriber which does not halt (whether prematurely by failure or intentionally) will eventually deliver every such obvent.

Certified: With such obvents, even if a subscriber temporarily disconnects or fails, it will eventually deliver the obvent.[2]

Totally ordered: Obvents can furthermore be notified in a total order to the subscribers: roughly spoken, two subscribers s_1 and s_2 which deliver two obvents o_1 and o_2 both deliver o_1 and o_2 in the same order (we also term this *subscriber-side order*).

FIFO ordered: Two obvents o_1 and o_2 that are published through the same object are delivered to all objects whose subscription matches both o_1 and o_2, and in the same order they were published (*publisher-side order*).

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

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

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

Timely: Similarly, certain obvents might "expire". In other terms, obvents can become obsolete.

These different semantics are not all mutually exclusive. For instance, obvents can be certified and have some notion of priority, or be certified and totally ordered at the same time. It appears that contradictions reside for instance between reliable and simultaneously timely limited obvents, as well as between orders (total, FIFO, or causal order) and priorities. In the above cases, the first type takes precedence (Figure 5.4 illustrates the dependencies between the different semantics).

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

Expressing Obvent Semantics

To avoid any complex negotiation of QoS semantics between participants, and since we abolish any first-class channel abstraction which could reflect QoS, such semantics

[2]We use here the terminology of [TIB99]. [OPSS93] refers to this as *guaranteed delivery.*

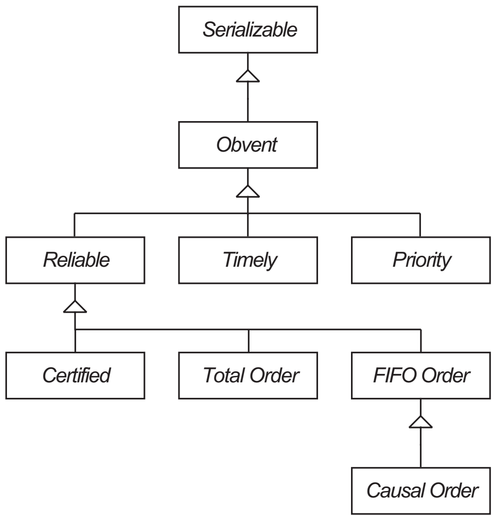


Figure 5.4: Dependencies Between Obvent Semantics

can be expressed by associating them with obvents, i.e., by making them become part of these obvents. Indeed, it makes most sense that every obvent reflects its semantics (which can be seen as a context), such that a correct handling of the obvent can be assured at every moment of the transfer.

Expressing QoS through subtyping. Since instances of an obvent type are bound to the same obvent source, they present the same characteristics, and the only (implicit) contracts between publishers and subscribers are these types. We have hence chosen to associate such characteristics with the obvent types. In other terms, we introduce different abstract obvent types, which can be subtyped by application-defined types to inherit their QoS. Figure 5.3 shows the Java types corresponding to the different delivery semantics outlined above.

Our DACE framework is constantly updated with new types representing concrete dissemination algorithms. The corresponding obvent types also subtype those basic obvent types of Figure 5.3 whose properties they inherit.

Multiple subtyping. Since several characteristics can be combined, this scheme can strongly benefit from a mechanism for expressing multiple subtyping. This is independent of whether it is assured through some form of (static) multiple inheritance (e.g., C++, Cecil, Eiffel), by subtyping abstract types (like interfaces in Java), or even through mixin inheritance (e.g., Flavors [Moo86], Ada). The term *multiple subtyping* here simply denotes the ability of expressing multiple specialization relationships.

5.3 Subscribing

As briefly illustrated in Section 5.1, we introduce a second primitive ,`subscribe`, to express a subscription.

5.3.1 Syntax

A subscription expression combines the subscription to a type T with (1) a closure declaration representing a filter, and (2) a second closure defining an obvent handler. The signature of the first closure can be simply outlined by the following:

$T \mapsto boolean$

Closures representing obvent handlers have a different syntax, since they need not return any value:

$T \mapsto void$

A subscription expression hence has the following syntax in Java (details are given in Figure 5.5):

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

Here *ObventType* represents a type which can be widened to the `Obvent` type, that is, *ObventType* is a special case of the *ClassOrInterfaceType* (§ 4.3 in [GJSB00]).

The creation of a subscription returns an object of type `Subscription` (cf. Figure 5.6). Such a handle uniquely identifies a subscription on a given host.

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

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

5.3.2 Obvent Handlers

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

Implementing Callbacks

In standard Java, and other languages lacking support for closures (or *higher order functions*), the execution of custom code respecting a given signature is often implemented by capturing one or several methods with required signatures in an abstract type. The application then provides a "callback object" as an instance of the application's own implementation of that abstract type. In Java, an interface

Primary: § 15.8
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
Type . `class`
`void` . `class`
`this`
ClassName . `this`
(*Expression*)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess
Subscription

***Subscription**:*
`subscribe` *SubscriptionDeclaration*

***SubscriptionDeclaration**:*
SubscriptionDeclarator FilterBody HandlerBody

***SubscriptionDeclarator**:*
(*SubscriptionFormalParameter*)

***SubscriptionFormalParameter**:*
ObventType Identifier

***FilterBody**:*
Block

***HandlerBody**:*
Block

***ObventType**:*
ClassOrInterfaceType

Figure 5.5: Precise Syntax of Subscription Expressions

```
package java.tps;

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

Figure 5.6: Subscriptions in Java$_{PS}$

implemented for callbacks, whose instances take the role of the observer according to the observer design pattern, is commonly called a *listener.* To enforce strong typing in our case, the type of the formal argument of a callback method in a listener must conform to the type of the obvents of interest. This can easily be achieved in languages which support a form of genericity, without generating specific listener types for every potential obvent type. This has been exploited in the library approach.

Regrouping Related Code

In any case, such a listener leads to isolating the obvent handling in a separate class, and hence potentially scattering the code related to a subscription throughout the entire program. The use of a closure on the other hand enables the regrouping of all code related to a subscription in a single succinct expression. If desired, the code for handling obvents *can* obviously still be isolated from the effective subscription expression, e.g., by simply forwarding the handled obvents to a previously instantiated class.

With the closures we introduce here silently, callback objects can be avoided, and there is no need for specific listeners or parametric polymorphism to avoid an important source of type errors, i.e., obvent delivery (cf. Section 3.6.1).

By viewing these closures as objects, the handlers take the role of the subscriber objects outlined in Section 3.2.

5.3.3 Filters

Akin to handlers, 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 conducted by the desire of confining the code for the filtering, while still "revealing" it.

Optimizable Filters

This way of expressing filters enables (1) the migration of the code to foreign hosts possibly entirely dedicated to filtering, as well as (2) the optimization of the filtering phase by delaying evaluation of filters and regrouping these in order to avoid redundancies in filters of different subscribers gathered on individual hosts.

The application context is one of the two main differences between our filters and *Application-Specific Handlers* (ASHs) [WEK97]. The latter filters similarly represent some form of message filters, yet are however more low-level and applied locally.

Deferring Evaluation

The second difference consists in the expressing of our obvent filters in the native Java language syntax, while ASHs promote a neutral specification language. In our case, this enables the static type-checking of these filters. Nevertheless, the evaluation of these filters is deferred, in a sense similar to the paradigm of deferred code evaluation underlying *two-level programming* [NN88], or generalized to more than two levels, *multi-stage programming*, as advocated by MetaML [TS97].

Deferred evaluation is in some sense already given by the closure paradigm. In our context, one could also refer to this way of proceeding as *deferred compilation*, since no compilation is immediately performed. However, as we will elucidate in Section 5.4.4, compilation, in the proper sense, might not be performed at all, by making use of dynamic invocations like in the case of our filter library discussed in the previous chapter.

5.3.4 Restrictions on Closures

Handlers and filters hence represent different forms of closures, with different semantics and restrictions.

"Local" Closures

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

"Distributed" Closures

Our "distributed" use of closures in the case of filters requires however even more restrictions.

Transparent filters. Any variable used in a filter might reference an object (which might reference an object, etc.) of a type which is not known on a host where that filter is evaluated, forcing the transfer of code. Similarly, any method invocation, whether performed on a variable or as a static call, might force the transfer of further code. In the case of Java, a class can be compiled despite the unavailability of the source code of all types it uses, making it very difficult to foresee the effects of calls to such classes.

The situation of missing classes is of course not impossible to handle (Section 3.4.1), and can occur, as a consequence of subtype substitutability, when deserializing obvents. E.g., an obvent attribute can contain an instance of a subtype of the static attribute type.

Pragmatic semantics. In the absence of a precise formalism, we have adopted a pragmatic approach to cutting down method invocations (including the use of constructors) to the essential ones. The goal is to avoid, as far as possible, filters using opaque code as well as types unknown on filtering hosts. The following restrictions are imposed on filters:

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

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

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

Loopholes. As illustration of the inherent difficulty of guaranteeing the location-independency of filters, consider the following case: as already mentioned in Section 4.5.1, Java includes reflective capabilities, which can be "misused" in several ways. In particular, any Java object gives access to a meta-object representing its class (`java.lang.Class`) through a `getClass()` method, similarly to Smalltalk, through which basically any class can be reached (invoked) by respecting the guidelines given above. We will come back to this particular loophole in Section 5.5.

5.3.5 Concurrency

Once an obvent has reached an address space hosting an interested subscriber, it is delivered by executing the handler. The handler can allow several levels of concurrency.[4]

Thread Policies

In Java$_{PS}$, we distinguish two main kinds of *thread policies*, leading to different levels of concurrency in obvent handlers.

[3]In terms of the different levels of expressiveness outlined in Section 3.5.3, this allows for accessors as expected results, and invocation arguments.

[4]This issue has not been discussed in the library approach.

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

Single-threading: A handler never processes more than one obvent at a time. In the case of ordered obvents, these semantics are required to not compromise the order defined by the underlying dissemination algorithm.

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

Expressing Thread Policies

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

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

However, to obtain more sophisticated concurrency control, or to involve the publish/subscribe system into concurrency issues with the goal of optimizing concurrency, thread policies should be made explicit. In our case, it seems most straightforward to express these through the subscription handle, since such an object uniquely defines a subscription. To control such parameters, corresponding methods are added to the `Subscription` type shown in Figure 5.6.

5.3.6 Managing Subscriptions

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

Activating a Subscription

Instead of defining specific language primitives for activating (deactivating) a subscription, we have chosen to provide these actions through method calls on subscription handlers.

A subscription is activated by a call to the `activate()` method on the corresponding subscription handle. An exception of type `CannotSubscribeException` is thrown by this method if the subscription cannot be issued, e.g., if the subscription is already activated.

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

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

Deactivating a Subscription

Similarly, the action of unsubscribing is expressed through a `deactivate()` method defined on subscription handles. This method can throw an exception of type `CannotUnsubscribeException`.

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

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

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

5.3.7 Programming the Stock Trade Example with Java$_{PS}$

To contrast the present language integration approach to TPS with the library approach outlined in the previous chapter, we return to the stock trade example presented in Section 4.6.

GDACs vs Java$_{PS}$ for Stock Trading

The only difference between the obvent types used here and the ones outlined in the previous relies on the fact that we introduce an explicit root type for all obvents in the case of Java$_{PS}$. The resulting `StockObvent` hence simply subtypes `Obvent` instead of `Serializable`.

```
public abstract class StockObvent implements Obvent {...}
```

Publishing Stock Quotes

Stock quotes can be published by writing something like the following:

```
StockMarket market = ...; /* reference to this market */
StockQuote q = new StockQuote("Telco Mobiles", 80, 100, 12373, market);
publish q;
```

This is indeed much simpler than publishing through a GDAC.

Subscribing to Stock Quotes

Akin, a subscription can be expressed rather easily. The subscription issued below covers the same set of obvents than the one presented in the context of the library approach:

```
final StockBroker broker = ...; /* this broker */
...
Subscription s = subscribe (StockQuote q)
  {
    return (q.getCompany().indexOf("Telco") != -1
            && q.cheaper(100.0));
  }
  {
    ...
    boolean bought =
      q.getMarket().buy(q.getCompany(), q.getPrice(),
                        q.getAmount(), broker);
    ...
  };
s.activate();
...
```

In terms of clarity, the approach used in the context of GDACs for the expression of subscriptions is in no case comparable to the present solution.

5.4 Translating Primitives

We have adopted the approach of transforming TPS-specific statements and expressions to standard Java code, more precisely, to method invocations. For illustration purposes, we describe the translation procedure following a form of heterogenous translation. The possibility of a more homogenous translation will be discussed in Section 5.5.3.

5.4.1 Typed Adapters, Listeners and Filters

To avoid making the Java virtual machine distribution-aware, we use adapters to encapsulate the entire communication middleware. These adapters mainly bridge the gap between serialized generic objects and strongly typed obvents. In contrast to the GDAC approach reported in the previous chapter however, adapters are here generated at compilation for any given obvent class C. Besides the corresponding `CAdapter` class encompassing code for publishing, and subscribing to, instances of C, a listener type `CSubscriber`, and a filter type `CLocalFilter` for local application (see Section 5.4.4) are generated. Similarly, to support subscriptions to abstract types, for any given abstract obvent type I (including abstract classes), a class `IAdapter` is generated with code for subscribing to instances of I, as well as a listener type `ISubscriber` and a filter type `ILocalFilter`. Figure 5.7 illustrates an adapter and the corresponding listener for a given obvent type T.

```
import java.tps.*;

public final class TAdapter {
  /* subscribing */
  public static Subscription subscribe(TLocalFilter l, TSubscriber s) {...}
  public static Subscription subscribe(RemoteFilter r, TSubscriber s) {...}
  /* publishing, only if T is a non-abstract class */
  public static void publish(T t) throws CannotPublishException {...}
  public static Class getType() { return T.class; }
  ...
}

public interface TSubscriber {
  /* deliver obvent */
  public void notify(T t);
}

public interface TLocalFilter {
  /* filter obvent */
  public boolean conforms(T t);
}
```

Figure 5.7: Types Generated for an Obvent Type T

5.4.2 Publishing

Since a published obvent is disseminated through the adapter for its dynamic type, which might be only known at runtime, a *PublishStatement* cannot be directly transformed to a call to a `publish()` method on the corresponding adapter class. Hence, we add a `publish()` method to the `Obvent` interface in Java (Figure 5.8), whose body is however automatically generated at compilation for each obvent class C in the application:

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

Accordingly, a *PublishStatement* expressing the publishing of an obvent o,

```
publish o;
```

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

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

Without this condition, any object declaring a parameterless `publish()` method could be accepted as argument to the `publish` primitive.

```
package java.tps;

import java.io.*;

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

/* for predefined types */
public final class ObventAdapter {
  public static Subscription subscribe(ObventLocalFilter l,
                                      ObventSubscriber s) {...}
  public static Subscription subscribe(RemoteFilter r,
                                      ObventSubscriber s) {...}
  public Class getType() { return Obvent.class; }
  ...
}
public interface ObventSubscriber {
  public void notify (Obvent o);
}
public interface ObventLocalFilter {
  public boolean conforms(Obvent o);
}

public final class ReliableAdapter {...}
...

/* remotely applied filters */
public class RemoteFilter {...}
...
```

Figure 5.8: Details of Class `Obvent`, and further Types in `java.tps`

5.4.3 Subscriptions

By similarly transforming subscriptions to invocations of (static) methods on the 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 one of the `subscribe()` methods in class `TAdapter` (possibly `ObventAdapter` or an adapter for any other predefined obvent type), as shown in Figure 5.7.[5]

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

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

Such an anonymous class implements the `TSubscriber` interface given in Figure 5.7:

```
new TSubscriber() {
  public void notify(T t) {
    /* handler */
  }
}
```

An anonymous class declaration, just like a closure can be passed as argument, here to the `subscribe()` method of the corresponding adapter.

5.4.4 Filters

The handling of filters represents the most complex task during compilation. A filter for a type T whose statements deviate from the guidelines which strongly enforce its mobility according to Section 5.3.4, is similarly transformed into an anonymous class representing a unary predicate.

Applying Filters Locally

Such a predicate is an instance of a specific filter type `TLocalFilter` inheriting from the `LocalFilter` type shown in Figure 5.8, and is applied locally. A subscription expression such as

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

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

```
TAdapter.subscribe(
  new TLocalFilter() {
    public boolean conforms(T t) {
      /* filter */
    }
    ...
  },
  new TSubscriber() {...}
)
```

[5]One can easily enforce a separation of different applications at the programming level, by prohibiting subscriptions to predefined abstract types.

Applying Filters Remotely

If the depicted restrictions are respected, an intermediate representation (similar to a first-class parse tree) of the filter is generated. In fact, most of the infrastructure devised for content-based filters in the library approach has been reused.

More specifically, our precompiler generates two tree-like constructs, which are more specific than for instance the parse trees used in Smalltalk (the structure of the program is reified by *nodes*: a *program node* contains *method node*s etc. [Riv96]). This information is stored in an instance of the `RemoteFilter` class (Figure 5.8).

Invocation tree: First, a representation of the invocations made in the filter is generated: the root represents the filtered obvent, and every intermediate node represents a method invocation. A leaf node stands for the outcome of a condition on the value obtained by applying the methods of the nodes on the path downto that leaf in a nested fashion (nodes can also represent attribute accesses). This tree conforms to the scheme presented in Section 3.5 for combining conditions, just like the sample invocation tree depicted in Figure 4.16.

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

5.4.5 Compilation

Our first implementation of Java$_{PS}$ was based on a precompiler [EGD01]. Like with most Java precompilers however, the developer ended up with two versions of the source code, namely the original one, and the translated code. The original Java compiler had to be run on the translated code, while the original one could not simply be discarded nor moved. Hence, two versions of the same code had to coexist, and these had to be managed in a way that prevented potential name conflicts.

To circumvent this deficiency, the precompiler was integrated with a Java compiler. More precisely, we have made use of JaCo [ZO01], an extensible Java compiler. Jaco made the integration of precompilation and compilation very easy.

This integration with the compiler made it difficult to translate our code to GDACs. Indeed, GDACs themselves already required a specific compiler, GJ, which was even extended to make up for the lack of runtime support.

5.5 Discussion

While devising our primitives for TPS in Java, we have considered many design and implementation alternatives. We discuss here the underlying issues through a subset of these alternatives.

5.5.1 Fork

This first alternative for the expression of TPS inside the Java language makes use of a `fork`-similar primitive for notification delivery.

Obvent Variable

A delivered obvent is assigned to a variable, and a block representing a handler is executed every time a new value is assigned to the variable:

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

This primitive could, in the case of Java, be implemented like the solution presented throughout this chapter. Yet, this syntax makes it difficult to express unsubscriptions: by the absence of a subscription handle (it competes for its place with the obvent variable), a subscription cannot be referred to from outside of the subscription expression. Unsubscriptions would have to be dealt with inside the handlers, which could be implemented through a parameterless unsubscription statement, or by having the handler return a boolean value after each obvent evaluation to signal whether the subscription is to be pursued. Either variant leads to a restrictive solution, where a subscription can *only* be cancelled after the next obvent has been delivered. While this can be desirable in many cases, it should not be the only possibility of cancelling a subscription. (In contrast, the solution adopted in Java$_{PS}$ allows this, yet does not force it.)

In contrast, this problem does not occur in the case of a "classic" `fork` primitive for spawning a new coroutine, since the execution of the block performed by the new coroutine takes place once only. Here, notifications are delivered continuously, leading to a repeated evaluation of the handler.

Filter

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

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

As explained previously in Section 3.1.2 however, filtering obvents by matching them against template objects offers only little expressiveness, and either violates encapsulation, or renders the matching opaque to the middleware, disabling performance optimizations targeted at avoiding redundancies in filters.

5.5.2 Callback

As outlined in Section 5.3.2, a very common way of implementing a callback in a language such as Java consists in constraining the application to provide a callback object implementing a given interface.

Implementing Listeners

This second considered alternative comes closer to the practices characteristic for nearly all Java API's for publish/subscribe engines, including GDACs. A specific listener like the `ObventSubscriber` (Figure 5.8) is in this case explicitly implemented and instantiated by the application to catch obvents:

```
class MySubscriber implements ObventSubscriber {
  public void notify(Obvent o) {
    StockQuote q = (StockQuote)o;
    System.out.print("Got offer: ");
    System.out.println(q.getPrice());
  }
}

ObventSubscriber sr = new mySubscriber();
Subscription s = subscribe (T t) { /* filter */ } sr;
```

Also, this approach could enable an easy expressing of thread policies: by subtyping specific abstract callback types, concrete subscribers could inherit threading policies. Multiple subtyping, exploited to express a limited form of QoS for obvents, could here be used again to express complex obvent handling semantics.

As shown by the above code extract however, the declared `notify()` method with its weakly typed argument strongly jeopardizes type safety.

Dynamic Overriding

A *dynamic overriding* of the `notify()` method, e.g., adding a variant with an argument type `StockQuote` in `MySubscriber` without implementing a specific listener `StockQuoteSubscriber`, would allow handlers for several obvent types to be provided by the same subscriber.

We have been initially attracted by the flexibility such a solution could offer by allowing handlers for several obvent types, possibly subtypes of the actually subscribed type(s), to be provided by the same subscriber.

Dynamic dispatching. This dynamic overriding of the `notify` method would however make it a so-called *multi-method.* Yet, *dispatching* (method selection) in Java, unlike CommonLisp or Cecil, does not support multi-methods. Dispatching in Java, similarly than in C++, offers *dynamic uni-dispatch*, i.e., the class of the object referenced by an invoked variable (representing its dynamic type) is determined at runtime, but only *static multi-dispatch.* This prevents a typed solution based on dynamic overriding of the `notify()` method described above.

There have been several approaches to overcoming Java's lack for multi-methods, ranging from an intensive and costly use of reflection [Cha98], using a precompiler [BC97], modifying the compiler [CLCM00], to extending the virtual machine [DLS$^+$01].

Double dispatch. A common workaround to this problem is also given by *double dispatch* [Ing86], which could however not be applied in this case, unless a specific subscriber type is again generated and implemented for every obvent type that a subscriber type handles, introducing a strict matching between handlers and handled types.

This again introduces a strict equivalence between formal arguments of handlers and the actually handled types. In our eyes, this contradicts the only valuable justification for an approach based on listeners, which is precisely the freedom of defining obvent handlers for *any* types.

Bottom line. We have indeed investigated a merging of the ideas advocated by the precompiler or compiler approaches mentioned above with the translation of our primitives, for instance by inserting code at the beginning of the `notify()` method for handling instances of type `Obvent` with explicit type checks (`instanceof`) and dispatches (with corresponding casts). Yet, we deem dynamic multi-dispatch a high price to pay if it is only introduced for the use with TPS primitives,[6] especially given the fact that it does not improve type safety.

Note furthermore that the scenario of multiple subscriptions of a same listener to the same type, or through subtyping related types, is not straightforward to handle: are the different filters combined, and is the same obvent delivered several times? These potential conflicts can be observed in any model which dynamically associates first-class entities (e.g., representing handlers or filters) in subscriptions (e.g., ECO, cf. Section 3.7.2).

5.5.3 A Homogenous Translation

A homogenous translation of TPS primitives can be a viable solution as well. The above heterogenous translation has in fact been driven mainly by early implementations, which relied on a *class-based* dissemination model [EG00], where performance could benefit from the static nature of adapters. With dissemination algorithms, such as the one we present in the next chapter however, the TPS middleware is pictured as a single channel, conveying objects of type `Obvent`.

[6]This same remark could be made regarding closures. To the application developer, closures do however less appear as such, since they are "inlined" in subscription expressions.

```
package java.tps;

public final class ObventAdapter {
  ...
  public static Subscription subscribe(ObventLocalFilter l,
                                       ObventSubscriber s,
                                       Class c)       {...}
  public static Subscription subscribe(RemoteFilter r,
                                       ObventSubscriber s,
                                       Class c)       {...}
  public static void publish(Obvent o) throws CannotPublishException {...}
  ...
}
```

Figure 5.9: Details of Class `ObventAdapter`

From Heterogenous to Homogenous

Here, the basic types `ObventAdapter`, `ObventSubscriber`, and `ObventLocalFilter` are sufficient for publishing and delivering obvents.

Nevertheless, the `ObventAdapter` type is augmented to reflect the dynamically typed nature of this translation. First, a `publish()` method is added, through which *all* obvents will be published (in contrast, in the heterogenous approach, nothing is published through the `ObventAdapter`, since it reflects an abstract type). Second, the `subscribe()` methods are augmented by a parameter representing a class meta-object, since the type of interest must be in any case advertised underneath to ensure a correct routing of obvents.

Publishing

In this case, a *PublishStatement* expressing the publishing of an `Obvent` *o*,

```
publish o;
```

can be simply transformed into a call to the `publish()` method in the `ObventAdapter` class, provided that *o*'s static type can be widened to `Obvent`.

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

Subscriptions

For subscriptions, consider the simple case where a local filter is generated from a subscription such as the following:

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

Such a subscription expression is transformed to the following call, which nevertheless provides type safety:

```
ObventAdapter.subscribe(
  new ObventLocalFilter() {
    public boolean conforms(Obvent o) {
      T t = (T)o;
      /* filter */
    }
  },
  new ObventSubscriber() {
    public void notify(Obvent o) {
      T t = (T)o;
      /* handler */
    }
  },
  T.class
)
```

Remotely applied filters, like in the heterogenous approach, and similarly to the filters used in the library approach, can be considered as untyped. Indeed, they contain merely dynamic type information, since they are not compiled and not statically evaluated.

5.5.4 Failures and Exceptions

Remote interactions involve different semantics and failure patterns than local ones. QoS, which allow to grasp the more complex semantics in remote interactions, have been discussed abundantly in the context of the present language integration approach to TPS. Failure patterns on the other hand can be reflected by exceptions, which in common middlewares are dealt with according to two diverging philosophies.

Hiding Distribution

In the previous library approach, subscribers are forced to deal with exceptions, by adding a method for exception handling to the listener type implemented by the application. In the current language integration we have considered two alternatives given by the rich exception handling features offered by the Java language.

In Java, `RuntimeExceptions` represent exceptions occurring during the "normal" operation of the Java virtual machine. They do not have to be declared in method signatures, nor do they have to be caught. The Java mapping for CORBA introduces communication failures as subtypes of such runtime exceptions, which consequently do not have to be caught. This gives the illusion that remote and local invocations can be viewed as equivalently behaving.

Embellishing Distribution

Interestingly, Java RMI does not promote its basic type for exceptions reflecting problems due to the remote nature of invocations, the `RemoteException`, as runtime exception, but instead forces every method in a remotely accessible interface to reflect the possibility of such an exception during invocation. Consequently, in every remote invocation, exceptions have to be dealt with explicitly.

We have pursued a similar approach to Java RMI, since we are convinced that it is a bad approach to try to hide this aspect of distribution (like certain others). We believe that adequate abstractions for distributed programming should behave much like a facial lifting for distribution, by making its appearance nicer, but that distribution has certain facets which simply have to be dealt with at some point.

Exception Handlers

Remember that in the GDAC approach, a consumer has to implement a method called `handleException()` for handling any kind of exceptions that could occur, and that the methods used for the actual interaction such as `add()`, or `contains()`, do not throw any exceptions, as opposed to the corresponding primitives in Java$_{PS}$. We believe that it is a better approach to reflect exceptions exactly where they can occur, as in Java$_{PS}$, but the approach chosen with the GDACs was motivated by two reasons, namely that (1) otherwise new exceptions would have been added to the exceptions thrown by methods inherited from the original collections, and that (2) the introduced exception handler could be used at the same time as an identifier for subscriptions, to avoid the problems described above when subscribing several times the same listener.

5.5.5 Structural Conformance

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

Reflection

A simple, though not very effective, form of structural conformance could be achieved in the case of a language integration, similarly to the library approach, through reflection. Java's introspection mechanisms enable the querying of arbitrary objects for their type, and also members, through the `getClass()` method:

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

With the above filter, *any* obvent type which implements a given method `getPrice()` (e.g., the `StockQuote` class) can be captured, and the following handler could similarly dynamically extract information from a conforming obvent.

Note that the self-describing events advocated by [OPSS93] and implemented by most existing engines aim precisely at making up for this lack in most programming languages.

Tuple Spaces: Back to the Roots

Another way of achieving structural equivalence could consist in coming back to the concept of tuples. In that sense, the `publish` primitive could be extended in order to accept any number of actual arguments:

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

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

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

which could all be used as subscription criteria by the filter, and could all be accessed by the handler. This requires however more complex filtering and an alternative way of expressing QoS.

Summary

Quite obviously, the presented extension of the Java language enables the writing of much more succinct and clear code for TPS than the library approach. Especially subscription patterns become intuitively understandable and also statically type-safe, by using “standard” Java syntax.

These benefits however come at a fairly high cost, namely closures, and a specific form of deferred code evaluation are introduced for the sole purpose of rendering subscription pattern expression more intuitive. Also, the proposed deep integration is not easily portable to other languages, since, besides serialization already presupposed in the previous library approach, also multiple subtyping is exploited for specifying a form of QoS. (We will come back to language mechanisms and their application in distributed programming in the concluding remarks.)

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

Chapter 6

hpmcast: A Dissemination Algorithm for TPS

In our DACE project, diverging requirements expressed through QoS are mainly explored by a variety of different delivery semantics implemented through different dissemination algorithms ranging from “classic” *Reliable Broadcast* [HT93], to new and original algorithms, like the broadcast algorithm we introduce in [EBGS01], and which ensures reliable delivery of events despite network failures.

While striving for strong scalability, we have invested considerable effort in exploring *probabilistic* (*gossip-based*) algorithms. These appear to be more adequate in the field of large scale event dissemination than traditional strongly reliable approaches like [HT93]. Basically, probabilistic algorithms trade the strong reliability guarantees against very good scalability properties, yet still achieve a “pretty good degree of reliability” [EKG01].

Until now, most work on gossip-based algorithms considers broadcasting information to all *participants* in a system, paying little or no attention to individual and dynamic requirements, as typically encountered in content-based dissemination.

We present here *Hierarchical Probabilistic Multicast* (*hpmcast* [EG01b]), a novel gossip-based algorithm which deals with the more complex case of multicasting an event to a subset of the system only. Requirements, such as limiting the consumption of local memory resources by view and message buffering, as well as exploiting locality (the proximity of participants) and redundancy (commonalities in interests of these participants), are all addressed.

Though *hpmcast* has been motivated by our specific context of TPS, it is general enough to be applied to any context in which a strongly scalable primitive for event, message, or information dissemination is required.

6.1 Background: Probabilistic Broadcast Algorithms

The achievement of strong reliability guarantees (in the sense of [HT93]) in practical distributed systems requires expensive mechanisms to detect missing messages and initiate retransmissions.

6.1.1 Reliability vs Scalability

Due to the overhead of message loss detection and reparation, algorithms offering such strong guarantees do not scale over a couple of hundred participants [PS97].

Network-Level Protocols

In [EGS00], we describe a simple publish/subscribe architecture based on IP Multicast. Such network-level protocols however have turned out to be insufficient: IP Multicast lacks any reliability guarantees, and so-called reliable protocols do not scale well. The well-known *Reliable Multicast Transport Protocol* (RMTP) [PSLB97] for instance generates a flood of positive acknowledgements from receivers, loading both the network and the sender, where these acknowledgements converge.[1] Moreover, such protocols hide any form of *membership* [ADKM92, MMSA91], making their exploitation more difficult with more dynamic dissemination (filtering).

Probabilistic Algorithms

Gossip, or *rumor mongering algorithms* [DGH$^+$87], are a class of *epidemiologic algorithms*, which have been introduced as an alternative to such reliable network-level broadcast protocols. They have first been developed for replicated database consistency management, and have been mainly motivated by the desire of trading the strong reliability guarantees offered by costly deterministic algorithms against weaker reliability guarantees, but in return obtaining very good scalability properties.

The analysis of such algorithms is usually based on stochastics similar to the theory of epidemics [Bai75], where the execution is broken down into steps. Probabilities are associated to these steps, and such algorithms are therefore sometimes also referred to as *probabilistic algorithms.*

Reliability Degree

The "degree of reliability" is typically expressed by a probability; like the probability 1-α of reaching *all* participants in the system for any given message, or by a probability 1-β of reaching *any* given participant with any given message. Ideally,

[1]Similarly, the scalability offered by other reliable network-level protocols, like *Reliable Multicast Protocol* (RMP) [WMK95], *Log-Based Receiver-Reliable Multicast* (LBRM) [HSC95], or *Scalable Reliable Multicast* (SRM) [FJM$^+$96] is not sufficient for many current application scenarios.

α and β are precisely quantifiable. A more precise measure, called Δ-*Reliability*, based on the distribution of the probability of reaching a fraction of participants, is given in [EKG01].

6.1.2 Basic Concepts

Decentralization is the key concept underlying the scalability properties of gossip-based broadcast algorithms, i.e., the overall load of (re)transmissions is reduced by decentralizing the effort. Participants are viewed as *peers*, symmetric in role, which are all equally eligible to forward information.[2] This makes gossip-based algorithms ideal candidates for systems with an underlying *peer-to-peer* model.

Parameters

More precisely, retransmissions are initiated in most gossip-based algorithms by having every participant periodically, i.e., every P ms (*step interval*), send information to a randomly chosen subset of participants inside the system (*gossip subset*). The size F of the subset is usually fixed, and is commonly called *fanout*. Gossip algorithms differ in the number of times the same information is gossiped. Every participant might gossip the same information the same number of times, meaning that the number of *repetitions* is fixed. Alternatively, the same information might be forwarded only once by a same participant, and the longest causal chain of message forwards can be limited by fixing the number of *hops* H (or *forwards*). Also, the number of rounds T (step intervals) that a message remains in the system can be limited.

Approaches

Gossiping techniques have been proposed in a broad spectrum of contexts. Consequently, these algorithms vary a great deal in their further characteristics.

Messages. Gossip-based algorithms differ in the kind of information that is shipped by gossiped messages (*gossips*). In early gossip algorithms, gossips reflect the sender's message buffer, including information about missing messages. Gossips have also been used to directly propagate the multicast payload (e.g., [EGH$^+$01]), like events in the case of TPS.

Interaction between peers. With the latter type of gossip-based algorithms, the interaction between peers invariably relies on *pushing* messages, e.g., events, from one participant to a set of neighbors.

[2] Note that the SRM protocol also relies on a peer-based approach. A retransmitted message is however rebroadcast to the entire system.

The former type of gossip-based algorithms, i.e., aiming at propagating *digests*, vary in the way participants react to incoming gossip messages. With a *gossiper-pull*, a gossip receiver retransmits missing messages to the gossip sender. With *gossiper-push*, a gossip target replies with a retransmission request to the gossip sender. The term *anti-entropy* is sometimes used to denote a combined push/pull scheme, i.e., a bidirectional updating [Gol92]. In database replication, gossiper-pull has been shown to converge faster [DGH$^+$87]. The same observation is made in the context of gossip-based broadcast algorithms, when a majority of participants have a message [SS00].

Faces of Scalability

Gossip-based approaches are said to be inherently scalable, meaning that the consumption of network resources (the amount of network messages necessary for successfully disseminating an application message) increases only slightly with an increasing system size. Scalability appears however under a variety of other faces, which can be devoted different priorities, depending on the context.

The case of membership. Most importantly in the context of TPS, implementations of content-based publish/subscribe have revealed the inherent difficulty of mapping individual and strongly dynamic requirements to a set of static groups [OAA$^+$00]: not *all* possible values for *all* attributes of events are known in advance, especially as new event types are added at runtime. When considering a broadcast group for every possible subset of participants of a system of size n, the views of an individual participant sum up to a total of $\sum_{m=1}^{n-1} \binom{n-1}{m} m = (n-1)2^{n-2}$ entries.[3] Since these membership views have to be managed explicitly, a further barrier to scalability is introduced. There have been indeed proposals on how to reduce the views of participants, however again without taking into account individual interests of these participants.

Interferences. Furthermore, the scalability of an algorithm can be limited by the size of message buffers, requiring subtle schemes for garbage collection. In general, the different faces of scalability are intermingled. As a first example, by highly loading the network, information can be spread quickly, reducing the size of buffers. As a second example, message buffers and data structures representing the system view compete for memory resources. Last but not least, when performing filtering to avoid sending events to participants which do not manifest any interests in these events, network resources are more wisely used, at the expense of processing power.

6.1.3 Related Gossip-Based Algorithms

Instead of presenting an exhaustive view of all work on gossip-based algorithms to date, we overview approaches that are closest to ours. These approaches are

[3]And this without counting the participant itself. Otherwise, the sum totals even to $n2^{n-1}$ entries.

discussed by pointing out the way they deal with the different aspects of scalability overviewed above.

Probabilistic Broadcast

With *Probabilistic Broadcast* (*pbcast* [BHO$^+$99]), Birman et al. have triggered a resurrection of gossip-based algorithms. *pbcast*, which is also called *Bimodal Multicast*,[4], relies on two phases: a "classic" best-effort multicast (e.g., IP Multicast) is used for a first rough dissemination of messages. A second phase assures reliability with a certain probability, by using a gossip-based retransmission: every participant in the system periodically gossips a digest of its received messages, and gossip receivers can solicit such messages from the sender if they have not received them previously.

Membership scalability. The membership problem is not dealt with in [BHO$^+$99], but the authors refer to a paper by van Renesse et al. which deals with failure detection based on gossips [vRMH98], while another paper describes *Capt'n Cook* [vR00], a gossip-based resource location algorithm for the Internet, which can in that sense be seen as a membership algorithm.

This also enables the reduction of the view of each individual participant: each participant has a precise view of its immediate neighbours, while the knowledge becomes less exhaustive at increasing "distance". The notion of distance is expressed in terms of host addresses (names). Capt'n Cook only considers the propagation of membership information.

The *Grid Box* [GvRB01] describes a more recent approach to arranging participants in a system according to a hierarchy, for the means of computing an *aggregate function* on outputs of all participants (e.g., sensor values). The hierarchy used is in essence the same as the one we will discuss later on, yet is applied in a very specific manner, by aggregating values at every level of the hierarchy.

Buffer scalability. The significant work accomplished at Cornell around gossiping techniques also includes efforts on how to enforce scalability in message buffering, by limiting the number of participants which store a same message [OvRBX99], or applying gossiping techniques to perform garbage collection [GHvR$^+$97].

These, as well as the membership and failure detection facets of scalability are all dealt with separately, proving their applicability to a wide range of algorithms (even algorithms which do not make use of gossiping techniques for the main spreading of the payload), yet only little to no information is given on the possibility and consequences of a cooperation in *pbcast*.

[4]We will however adhere to the first name, to comply with the taxonomy adopted in Section 2.1.4.

Reliable Probabilistic Broadcast

Reliable Probabilistic Broadcast (*rpbcast* [SS00]), an algorithm developed by IBM in the context of the Gryphon project, is strongly inspired by *pbcast*. There are two main differences. First, while *pbcast* has been originally described as using gossiper-push, *rpbcast* uses a pull scheme for its faster convergence. Second, and more important, *rpbcast* adds a third phase to achieve strong reliability. The system is instrumented with *loggers*, which log messages on stable storage. These are consulted whenever the two initial phases fail in providing some participant with a *relevant*[5] message. [SS00] does not give hints on membership management, nor on message buffering.

Directional Gossip

Directional Gossip [LM99] is an algorithm especially targeted at wide area networks, developed at the University of San Diego. By taking into account the topology of the network and the current participants, optimizations are performed. More precisely, a *weight* is computed for each neighbour node, representing the connectivity of that given node. The larger the weight of a node, the more possibilities exist for it to be infected by any node. The algorithm applies a simple heuristic, which consists in choosing nodes with higher weights with a smaller probability than nodes with smaller weights, reducing the number of redundant sends.

Membership scalability. The algorithm hence supposes that not all participants are connected, or rather, know each other. This implies partial views, and in practice, a single *gossip server* is assumed per LAN which acts as a bridge to other LANs. This however leads to a *static hierarchy*, in which the failure of a gossip server can isolate several participants from the remaining system.

Determinism. An approach to analyzing the performance achieved when every participant attempts to infect a deterministically determined subset of the system, involves the same authors [LMM00]. Subsets are established through a graph connecting the participants, called *Harary* graph, leading to a flooding of the system over such a graph. The introduced determinism, as intuition suggests, reduces the number of message sends. However, the reliability of the gossip-based algorithm used for comparison appears to degrade slightly more gracefully with an increasing number of participant failures, and the establishment of the connections according to the Harary graph introduces an important overhead.

[5]Similar to other algorithms, an upcall to the application determines how much effort is deemed suitable to recover a missed message [ORO00].

Lightweight Probabilistic Broadcast

Lightweight Probabilistic Broadcast (*lpbcast* [EGH$^+$01]) is a probabilistic broadcast algorithm developed in our lab. *lpbcast* adds an inherent notion of memory consumption scalability to the notion of network consumption scalability primarily targeted by gossip-based algorithms.

Probabilistic membership. In contrast to the deterministic hierarchical membership approaches in Directional Gossip or Capt'n Cook, *lpbcast* represents a probabilistic approach to membership: each participant has a *random partial view* of the system. *lpbcast* is lightweight in the sense that it consumes little resources in terms of memory and requires no dedicated messages for membership management: gossips are used to disseminate the payload (i.e., events) and to propagate digests of received events, but also to propagate membership information. The analysis presented in [EGH$^+$01] includes all of these aspects.

Probabilistic buffering. The basic *lpbcast* algorithm furthermore also buffers events in a fully probabilistic sense. To respect a maximum buffer size, every participant only buffers a random subset of the events gossiped in the system. This results in an effect similar to the one targeted by [OvRBX99], namely consisting in buffering individual messages only on a subset of the system

Optimizations for *lpbcast*, such as trying to "force" a more uniform distribution of the individual views, or prioritizing the buffering of more recent events, have been proposed in [KGHK01].

6.2 From Broadcast to Multicast

A broadcast algorithm can be obviously used to multicast events. We depict two gossip-based broadcast algorithms used to achieve multicasting of events, and outline the respective limitations of these approaches, leading to a more consolidated algorithm presented in the following sections.

6.2.1 Broadcast with Receiver Filtering

A pragmatic way of multicasting information inside a system subset consists in broadcasting within the entire system and filtering upon reception of events, i.e., an event is delivered to the application iff that participant is interested in that given event.

rfpbcast Algorithm

Figure 6.1 outlines a modified probabilistic broadcast algorithm called here *Receiver Filtering Probabilistic Broadcast* (*rfpbcast*). Every participant, similarly to a probabilistic broadcast, periodically (every P milliseconds) gossips to a randomly chosen subset of the system. In our context, a gossiper forwards every buffered event to a randomly chosen subset of size F of the system.

When receiving an event, a participant only delivers that event if it effectively is of interest for it.[6] This is represented at Line 16 of Figure 6.1 through the $\triangleleft$ operator indicating whether a given event is of interest for a certain participant. This can be viewed as evaluating the participant's subscription pattern for the considered event: *event* $\triangleleft$ *participant* returns *true* iff *participant* is interested in *event*.

Executed by participant$_i$

```
 1: initialization
 2: view
 3: gossips ← ∅

 4: task GOSSIP                                        {every P milliseconds}
 5:   for all (event, round) ∈ gossips do
 6:     if round < T then                              {not too many rounds}
 7:       round ← round + 1
 8:       dests ← ∅
 9:       repeat F times                               {choose potential destinations}
10:         dest ← RANDOM(view - dests)
11:         dests ← dests ∪ {dest}
12:         SEND(event, round) to dest

13: upon RECEIVE(event, round): do
14:   if ∄ (event, ...) ∈ gossips then                 {buffer the gossip and deliver it}
15:     gossips ← gossips ∪ {(event, round)}
16:     if event ◁ participant_i then
17:       RFPDELIVER(event)

18: upon RFPBCAST(event): do
19:   gossips ← gossips ∪ {(event, 0)}

20: function RANDOM(from)                              {choose random participants}
21:   return dest ∈ from
```

Figure 6.1: Receiver Filtering Broadcast Algorithm

Analysis

For our formal analysis we consider a system composed of n participants, and we observe the propagation of a single event notification. We assume that the composition of the system does not vary during the run (consequently n is constant). According to the terminology applied in epidemiology, a participant which has delivered a given notification will be termed *infected*, otherwise *susceptible*.

Assumptions. The stochastic analysis presented below is based on the assumption that participants gossip in synchronous rounds, and there is an upper bound on

[6]This is equivalent to performing the filtering in a higher layer, possibly in the application itself.

the network latency which is smaller than a gossip period P.[7] P is furthermore constant and identical for each participant, just like the fanout $F < n$. We assume furthermore that failures are stochastically independent. The probability of a network message loss does not exceed a predefined $\epsilon > 0$, and the number of participant crashes in a run does not exceed $f < n$. The probability of a participant crash during a run is thus given by $\tau = f \ / \ n$. We do not take into account the recovery of crashed participants, nor do we consider Byzantine (or arbitrary) failures. At each round, we suppose that each participant has a complete view of the system.

Number of infected participants. The analysis presented resembles the analysis applied to *pbcast* in [BHO+99] and *lpbcast* in [EGH+01]. The probability p that a given gossiped event is received by a given participant, is given as a conjunction of several conditions, namely that (1) the considered participant is effectively chosen as target, (2) the gossiped event is not lost in transit, and (3), the target participant does not crash.

$$p(n, F) = \left(\frac{F}{n-1}\right) (1-\epsilon)\ (1-\tau) \tag{6.1}$$

We denote the number of participants infected with a given event at round t as s_t, $1 \leq s_t \leq n$. Note that when the event is injected into the system at round $t = 0$, we have $s_t = 1$.

Accordingly, $q(n, F) = 1 - p(n, F)$ represents the probability that a given participant is *not* reached by a given infected participant. Given a number j of currently infected participants, we are now able to define the probability that exactly k participants will be infected at the next round ($k - j$ susceptible participants are infected during the current round). The resulting homogenous Markov chain is characterized by the following probability p_{jk} of transiting from state j to state k ($j > 1$, $1 \leq k \leq n$):

$$\begin{aligned} p_{jk}(n, F) &= P(n, F)[s_{t+1} = k | s_t = j] \\ &= \begin{cases} \binom{n-j}{k-j} (1 - q(n, F)^j)^{k-j} q(n, F)^{j\ (n-k)} & j \leq k \\ 0 & j > k \end{cases} \end{aligned} \tag{6.2}$$

The distribution of s_t can then be computed recursively ($1 \leq k \leq n$). In summary:

$$P(n, F)[s_t = k] = \begin{cases} 1 & t = 0, k = 1 \\ 0 & t = 0, k > 1 \\ \sum_{j=k/(1+F)}^{k} P(n, F)[s_{t-1} = j] p_{jk}(n, F) & t \geq 1 \end{cases} \tag{6.3}$$

[7] This analysis does not rely on the assumption that the underlying system is synchronous, nor does the algorithm force the system to behave so.

Expected number of rounds. In the *rfpbcast* algorithm, there are still two undefined parameters, which are the fanout F, and the number of rounds T. According to Pittel [Pit87], the total number of rounds necessary to infect an entire system of size n (large), in which every infected participant tries to infect $F > 0$ other participants, is given by the following expression:

$$\begin{aligned} & log_{\ F+1}\ n + \frac{1}{F}\ log\ n + c + O(1)\ = \\ & log\ n \left(\frac{1}{F} + \frac{1}{log\ (F+1)}\right) + c + O(1) \end{aligned} \tag{6.4}$$

By fixing either F or T, the other value can be computed based on this expression, or if more detailed information is required, through the above Markov chain.

However, the model in [Pit87] does not consider the possibility of losing events between a gossiper and a (potential) destination. In our case, only $F\ (1-\epsilon)\ (1-\tau)$ participants are expected to be infected at a given round by a gossiper, leading to the following expression:

$$T(n, F)\ =\ log\ n \left(\frac{1}{F\ (1-\epsilon)\ (1-\tau)} + \frac{1}{log\ (F\ (1-\epsilon)\ (1-\tau) + 1)}\right) + c + O(1) \tag{6.5}$$

Note that it has been shown in [KMG00] that, when limiting the number of repetitions to 1 (every participant forwards a given event at most once), choosing the natural logarithm of the system size as value for F brings the probability of reaching all participants very close to 1. Though in our model the number of gossips is not limited through the number of repetitions, but through the maximum number of rounds that an event can spend in the system, a logarithmic value could reflect a reasonable fanout.

6.2.2 Sender Filtering

A first, very strong limitation of the above *rfpcast* algorithm appears immediately. As reflected through the analysis, a gossiped event is sent to every participant, regardless of whether it is effectively interested in that event. Accordingly, especially with events which are of interest for only a small fraction of the system, there is a high waste of network bandwidth and also local memory for buffering.

To avoid sending an event to a participant for which that event is irrelevant, the following modified broadcast algorithm (*pmcast*) performs the filtering before sending. It can be seen as a *genuine multicast* ([GS01]) algorithm in the sense that events are only received by interested participants, and only these participants are involved in the algorithm.

pmcast Algorithm

Similarly to the previous *rfpbcast* algorithm, every participant periodically gossips every event in its buffer to a subset of participants in the system. In this *Probabilistic Multicast* (*pmcast*) algorithm (Figure 6.2) however, after picking a random subset of size F of the system, the set of destinations is further restricted to the subset of participants effectively interested in the considered event.

Note that no filtering is necessary at reception, since no spurious event is sent to any participant.

```
Executed by participant_i

initialization
view
gossips ← ∅

task GOSSIP                                         {every P milliseconds}
  for all (event, round) ∈ gossips do
    if round < T then                   {limit the time an event spends in the system}
      round ← round + 1
      dests ← ∅
      repeat F times                                {choose potential destinations}
        dest ← RANDOM(view - dests)
        dests ← dests ∪ {dest}
        if event ⊲ dest then
          SEND(event, round) to dest

upon RECEIVE(event, round): do
  if ∄ (event, ...) ∈ gossips then                  {buffer the gossip and deliver it}
    gossips ← gossips ∪ {(event, round)}
    PDELIVER(event)

upon PMCAST(event): do
  gossips ← gossips ∪ {(event, 0)}

function RANDOM(from)                               {choose random participants}
  return dest ∈ from
```

Figure 6.2: Probabilistic Multicast Algorithm

Analysis

The fraction of the system which is effectively interested in an observed event is of primary importance for the analysis. We can represent the size of the interested subset as $n\ p_1$, where p_1 is the probability that a given participant is interested in the event. In other terms, when considering that n_1 participants among n are interested in a particular obvent, then $p_1 = \frac{n_1}{n}$.

Number of infected participants. The effective expected number of participants that are gossiped to at each round by an infected participant is $F\ p_1$ (without considering network message loss and participant failures). The expression for p (the probability that a gossip reaches a given participant), is hence given in this case as follows:

$$\begin{aligned} p(n\ p_1, F\ p_1) =&\ \left(\frac{F\ p_1}{n\ p_1 - 1}\right)(1-\epsilon)\ (1-\tau) \\ q(n\ p_1, F\ p_1) =&\ 1 - p(n\ p_1, F\ p_1) \end{aligned} \tag{6.6}$$

This is similar to gossiping in an effective system of size $n\ p_1$, however with a fanout of only $p_1 F$. The resulting Markov chain is characterized by the probability p_{jk} of transiting from state j to state k, as follows:

$$\begin{aligned} p_{jk}(n\ p_1, F\ p_1) &= P(n\ p_1, F\ p_1)[s_{t+1} = k | s_t = j] \\ &= \begin{cases} \binom{n\ p_1 - j}{k-j}(1 - q(n\ p_1, F\ p_1)^j)^{k-j} q(n\ p_1, F\ p_1)^{j(n\ p_1 - k)} & j \leq k \\ 0 & j > k \end{cases} \end{aligned} \tag{6.7}$$

And the distribution of s_t can then again be computed recursively. In summary:

$$\begin{aligned} P(n\ p_1, F\ p_1)[s_t = k] \ = & \\ & \begin{cases} 1 & t = 0, k = 1 \\ 0 & t = 0, k > 1 \\ \sum_{j=k/(1+F)}^{k} P(n\ p_1, F\ p_1)[s_{t-1} = j] p_{jk}(n\ p_1, F\ p_1) & t \geq 1 \end{cases} \end{aligned} \tag{6.8}$$

Expected number of rounds. Similarly, the expression for the expected number of rounds above (Equation 6.5) can be adapted to reflect the sender filtering.

$$\begin{aligned} T(n\ p_1, F\ p_1) \ = & \\ & log\ n\ p_1 \left(\frac{1}{F\ p_1(1-\epsilon)(1-\tau)} + \frac{1}{log\ (F\ p_1(1-\epsilon)(1-\tau)+1)}\right) + c + O(1) \end{aligned} \tag{6.9}$$

Yet, since Pittel's formula is valid for large systems, this formula only offers useful results as long as $n\ p_1$ is still large.

6.3 Overview of Hierarchical Probabilistic Multicast

The above *pmcast* algorithm is indeed smarter than the *rfpbcast* algorithm, yet still presents an important number of sensible limitations. We discuss these and overview how our Hierarchical Probabilistic Multicast (*hpmcast*) repairs these lacks.

6.3.1 Membership Scalability

In both modified broadcast algorithms, every participant has a “full” view of the system, i.e., every participant knows every other participant. As already pointed out in [EGH$^+$01], this can become a severe barrier for scalability as the system grows in size.

In order to reduce the amount of membership knowledge maintained at each participant, a participant should only know a subset of the system. The individual subsets known by the participants should nevertheless ensure two properties, namely (1) that every participant is known by several others (for reliability), and (2) that no participant knows all participants (for scalability).

Random Approach

A random subset as chosen in *lpbcast* also ensures membership scalability, and can be put to work in a way that, with a uniform distribution of knowledge, the propagation of information is virtually not impacted by the reduction of the amount of membership knowledge. This approach applies well to broadcast, but gives less good results in a practical multicast setting, especially if the exploiting of locality and redundancy is desired.

Hierarchical Approach

In contrast, *hpmcast* is based on a hierarchical disposition of participants bearing strong resemblances with the Capt’n Cook and the Grid Box approaches. The extent of interactions between participants depends on their “distance”, but the hierarchical membership is used to multicast events inside the system. Participants have a complete knowledge of their respective immediate neighbours, but only a decreasing knowledge about more “distant” participants.

To that end, a participant can *represent* a *subnetwork*, or *subsystem*, for participants outside of the respective subsystem. In other terms, a participant outside of a subsystem, yet who knows that subsystem, will only know such representing participants, called *delegates*. Among the delegates for neighbor subsystems, again a set of delegates are chosen recursively, giving rise to a hierarchy of several (l) levels.

Delegates that appear at a high level in the hierarchy are known by more participants in the system than lower-level participants. Nevertheless, all participants have membership views of comparable sizes, since every participant has a view of its subsystem for every level. Also, a participant which is elected as delegate for a given hierarchy level still remains visible in views of its lower level subsystems.

6.3.2 Locality

Another limitation of the two previous algorithms (*rfpbcast* and *pmcast*) is that they do not take *locality* into account, i.e., a participant randomly picks destinations regardless of its "distance" to those participants. Far away participants are chosen with the same probability than close neighbors. It seems more adequate to aim first a, rough and wide distribution of events, before attempting a more complete and local dissemination.

Dissemination of events inside our hierarchy follows a *level-wise dissemination*, i.e., the highest level of the hierarchy is first infected, and then the following levels are successively infected in the order of their depth. This approach is opposed to the Grid Box, where however the goal is different; a global function has to be applied to values collected from all participants. In contrast, when multicasting inside the hierarchy, a value originates from a single participant and is propagated as such.[8]

Since higher levels regroup participants representing various distant subsystems, a level-wise gossiping ensures that events are in a first step spread coarsely, and that a finer coverage of the individual subsystems takes place successively. Only a "reasonable" number of sends between distant participants takes place, given by the number of gossip rounds expected to infect the corresponding level of the hierarchy. Once sparsely spread, an event is only more sent between more local participants.

6.3.3 Redundancy

Furthermore, in the *pmcast* algorithm, every participant stores every other participant's individual interests, and filtering is made independently for every chosen participant. Significant performance optimizations, like the exploiting of redundancies of individual subscription patterns, as proposed in [ASS$^+$99], cannot be applied.

The higher the level at which a delegate can be found in the hierarchy, the more participants that delegate represents. Accordingly, the delegate manifests interest in any event that is of interest for any of the participants it represents. Or, in the terminology of TPS, its subscription pattern, from the perspective of another participant, is a compound pattern created from the subscription patterns of the participants it represents. Redundancy of these individual patterns can be exploited, by creating such condensed patterns which avoid redundancies between the individual patterns.

Observe however that redundancy competes to some extent with locality: geographically "close" neighbors do not necessarily present "close" interests, and any scheme relying on one of these two notions of proximity might rule out the exploiting of the other notion.

6.3.4 Garbage Collection

Most dissemination algorithms apply a combination of *acknowledgements* (*acks*) and *negative acknowledgements* (*nacks*) to verify the stability of a given message and to perform garbage collection. Such schemes, as well as more sophisticated schemes (cf. [OvRBX99, GHvR$^+$97] for *pbcast*, or [KGHK01] for *lpbcast*), rely on unique

[8]It would be very interesting to exploit this aggregate function to reflect event correlation (see Section 2.4.4).

message identifiers. This again applies well to the case of broadcast, but is less straightforward to apply to a multicast setting, where a given multicast event is only significant for a subset of participants, and this significance can only be verified through the event itself, not through an identifier.

On the other hand, it is difficult to perform garbage collection by statically limiting the number of repetitions, forwards or hops. Indeed, as shown by Equation 6.9 above, parameters F and T are related, yet in a way depending on the fraction of interested participants (reflected by p_1), which is individual for every considered event. In fact, according to Equation 6.9, the number of rounds necessary to infect all interested participants increases as the number of these interested participants decreases. By fixing the number of rounds that an event can spend in the system, one ends up with considerably weaker reliability for events with a small "audience". While this can indeed make sense in specific contexts, we view this rather as an undesirable property.

We have hence chosen to integrate garbage collection into the multicast algorithm, by limiting the number of rounds that an event remains in the system. In fact, the expected number of rounds necessary to infect all participants in a subset of the system, as well as parameters of the system can be approximated. Inherently limiting this way the life-time of an event applies naturally, since gossips are used primarily to transport events, and not to exchange message identifiers aiming at detecting message stability. The feasibility of limiting the lifetime of gossiped events a priori, i.e., renouncing to any explicit garbage collection algorithm, has been illustrated by *lpbcast*.

6.4 Hierarchical Membership

This section describes a precise model of our hierarchy, while the corresponding membership management is informally described. We elucidate how this hierarchy, besides reducing the memory resource consumption of the membership view, also offers a nice compromise between locality and redundancy.

6.4.1 Model

As elucidated above, *hpmcast* is based on a hierarchical membership, where the knowledge that an individual participant has about other participants decreases with the "distance" from these participants.

Addresses

Before going further into the description of the membership, we require a definition of the notion of "distance" between two participants. An approach could consist in using an *average communication delay* between two participants as a measure for their distance. However, since we are considering asynchronous systems, such values are difficult to determine. We will thus base the decision of which participants out of a subsystem are to be considered as neighbors, and also as prioritized (to become delegates), on the *addresses* of those participants, more precisely, on the distances between them.

This notion of "distance" can be approximated by network addresses, but can as well be simulated by associating *logical* addresses with participants. Irrespective of how these addresses are determined, we will in the following simply consider such addresses as sequences of values, of the following form:

$$\begin{gathered} x(l-1).\cdots.x(0), \\ \forall i\ 0 \leq i \leq l-1, 0 \leq x(i) \leq a_i - 1 \end{gathered} \tag{6.10}$$

The total number of different addresses and thus the maximum number of participants is given by

$$\prod_{0 \leq i \leq l-1} a_i \tag{6.11}$$

Though participants in the sense of TPS can be colocated on the same host, or even in the same process, we will consider here for the sake of simplicity that there is one participant per host. Indeed, the last components of an address could easily be used to express a port number. To cover all possible IP addresses for instance, one could choose $l = 4$ and $a_i = 2^8 = 256\ \forall i$, or $l = 11$ and $a_i = 2^4 = 16\ \forall i$ to include $2^{12} = 4096$ port numbers.[9] DNS addressed would have to be inversed.

Branch Addresses

We call a "partial" address, like $x(l-1).\cdots.x(i)$ ($0 < i \leq l-1$; $\emptyset$ for $i = l$) a *branch address* of level i. Such a branch address denotes a subsystem or subnetwork. All participants sharing a branch address belong to the corresponding subsystem. In other terms, there might be several addresses $x(i-1).\cdots.x(0)$ that can be appended to a same branch address to denote a concrete participant.

The distance between two addresses is expressed based on this notion. In fact, the distance between two participants is equal to the level of their longest common branch address, e.g., if two participants p_1 and p_2 share a branch address of level i, then they are said to be at a distance of i. Also, they belong to the same subsystem of level i of the hierarchy. A distance of 0 would mean that the two participants share the same address and are thus equivalent.

[9]Note that certain IP addresses and port numbers are reserved for special purposes, e.g., IP addresses 224.0.0.0 to 239.255.255.255 are reserved for IP Multicast groups. Those exceptions will not be discussed here.

Electing Delegates

All participants which share a given branch address $x_0 = x_0(l-1).\cdots.x_0(1)$ form a group of level 1. The number of such participants (at the moment of observation) is denoted $|x_0(l-1).\cdots.x_0(1)|$. Quite obviously, for any given branch address $x(l-1).\cdots.x(1)$, $|x(l-1).\cdots.x(1)| \leq a_0$.

Of the $|x_0|$ participants, R delegates are chosen deterministically by all participants sharing x_0, e.g., by taking the R participants with the smallest addresses (we assume that $\forall\ x(l-1).\cdots.x(1)$, $|x(l-1).\cdots.x(1)| \geq R$, i.e., every populated system of level 1 contains at least R participants).

Figure 6.3 illustrates a simple example. R represents a *redundancy factor*, which however has no relationship with the notion of redundancy observed in subscription patterns. R represents the number of delegates that are elected to represent a subsystem, and is best chosen such that $R > 1$, in order to improve the reliability of the membership: with $R = 1$, the hierarchy is very sensitive to crash failures of individual participants, leading to an increased risk of a partitioned membership.

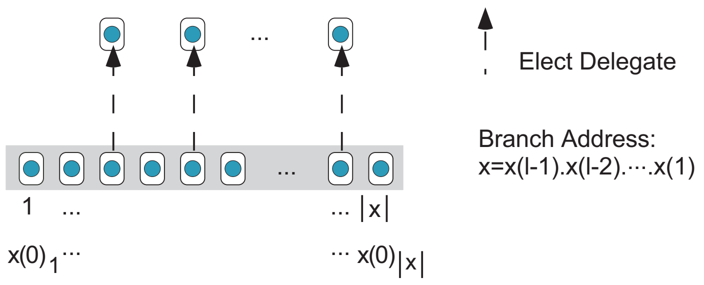


Figure 6.3: Electing Delegates for a Group of Level 1

In general, $\forall x = x(l-1).\cdots.x(i), |x|$ represents the number of different $x(i-1)$ that can be appended to x to denote a legal branch address, or in other words, the number of populated subsystems of x. Quite obviously, for any such branch address $x = x(l-1).\cdots.x(i)$, $|x| \leq a_{i-1}$.

Constructing a hierarchy. Together with R delegates for each other of the $|x_0(l-1).\cdots.x_0(2)|$ neighbor trees, the R delegates of $|x_0|$ form a group of level 2.

Recursively, any branch address $x_0(l-1).\cdots.x_0(i)$ $(l > i > 0)$ is shared by altogether $|x_0(l-1).\cdots.x_0(i)| \leq a_{i-1}$ subtrees with a different $x(i-1)$, each represented by R delegates. Together, these form a group of level i. (Figure 6.4). At the highest level (l), there are $|\emptyset| = |x_0(l-1).\cdots.x_0(i)|_{i=l}$ subtrees.

Remember that by promoting a participant as delegate, that participant's knowledge does not increase. This merely means that it is known by more participants, not that itself will have to know more participants. Consequently, a participant which appears as delegate of level i thus also appears as delegate of all levels i‘, such that $1 \leq i' < i$.

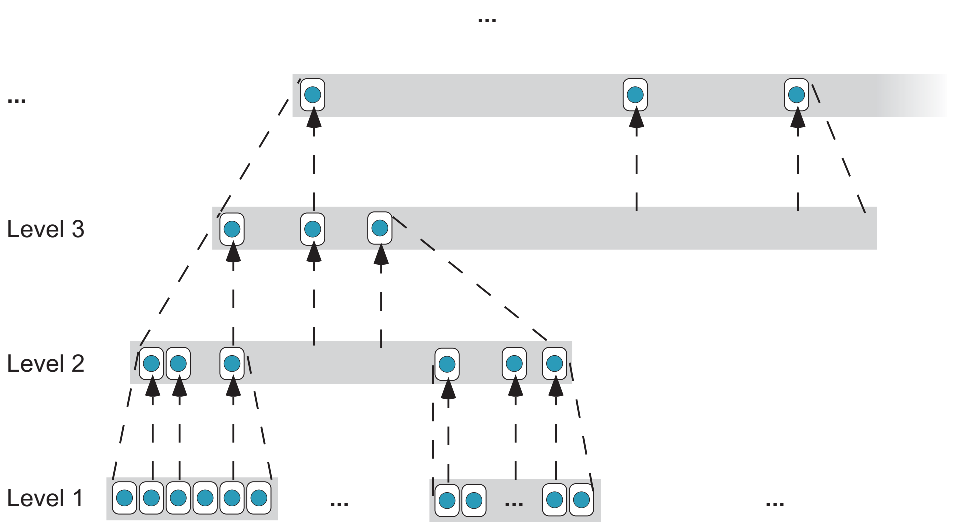


Figure 6.4: Electing Delegates Recursively ($R = 3$)

Individual knowledge. A given participant with address $x_0 = x_0(l-1).\cdots.x_0(0)$ knows for each of its branch addresses $x_0(l-1).\cdots.x_0(i)$ $(1 \leq i \leq l)$ all the $|x_0(l-1).\cdots.x_0(i)|$ different subtrees, and for each of those subtrees R delegates. Furthermore, for level 1, it knows all $|x_0(l-1).\cdots.x_0(1)|$ participants.

Thus, the total number of participants known by participant $x_0(l-1).\cdots.x_0(0)$ is

$$|x_0(l-1).\cdots.x_0(1)| + \sum_{i=2}^{l} R\,|x_0(l-1).\cdots.x_0(i)| \tag{6.12}$$

where a delegate of level i is also taken into account at any level below i. Also, the knowledge of the participant itself is considered (depending on the level of the considered participant, between 1 and l occurrences.)

Joint interest. The total number of participants that a delegate at level i with address $x_0(l-1).\cdots.x_0(0)$ represents is given by

$$\parallel x_0(l-1).\cdots.x_0(i) \parallel \;=\; \sum_{x(i-1).\cdots.x(1)} |x_0(l-1).\cdots.x_0(i).x(i-1).\cdots.x(1)| \tag{6.13}$$

Hence, when considering that every such participant is expected to be interested in a given event with p_1, the delegate itself, on behalf of these participants, is interested in any event with a probability

$$p_i = 1 - (1 - p_1)^{\|x_0(l-1).\cdots.x_0(i)\|} \tag{6.14}$$

6.4.2 Membership Management

Due to the inherent complexity of the membership algorithm, we present the way the membership is managed more informally.

The membership views are exchanged between participants through gossips, i.e., they are piggybacked by event gossips, except in the absence of events (dedicated gossips are used in this case). Hence, every participant periodically sends information about a random level of its view of the hierarchy to a subset of the system.

Propagating Information

More precisely, each participant maintains a table for each level, representing the participant's view of that level. Membership information updating is based on gossip-pull. To that end, every line in every table has a timestamp associated. This represents the last time the corresponding line of the table was updated. Periodically, a participant randomly selects participants of a hierarchy level and gossips to those delegates. A gossip carries a list of tuples (line, timestamp) for every line in every table. The receiver compares all the timestamps to its own timestamps, and updates the gossiper for all lines in which the gossiper's timestamps are smaller.

As explained above, such membership information can be piggybacked when gossiping events, or in the absence of such events, can be propagated with dedicated gossips. Similarly, as we will elucidate in Section 6.5.1, other gossips can be used to piggyback information. In some cases, this can even lead to a bidirectional updating as in anti-entropy (e.g., [Gol92]).

Joining. When a participant decides to join, it needs to know at least one participant. That participant contacts the "lowest" delegates it knows that the new participant will have. This is made recursively, until the immediate delegates of the new participant have been contacted. Hence, if the becoming participant contacts a "close" participant (if there are any), this procedure completes faster and induces less overhead.

Once neighbors (lowest-level neighbors of the becoming participant) have been contacted, these transmit their view of the system to the new participant.

The hierarchy is hence constructed stepwise, by adding one participant after another. An a priori knowledge of the approximate size of the hierarchy can in that sense be very useful to adjust parameters of the hierarchy, such as its depth: as we will show in the following sections, this parameter indeed has an impact on the performance of the system.

Leaving and Failures. The same lowest-level neighbors are also involved when a participant leaves. A participant wishing to leave will send a message to a subset of its lowest-level neighbors. These will remove the leaving participant from their views, and this information will successively propagate throughout the system through subsequent gossips.

For the purpose of detecting the failure of participants, every participant keeps track of the last time it was contacted by its lowest-level participants. This implements a simple form of failure detection, and makes sense at the lowest level, since such neighbors are supposed to be "close". The reduced average network latency makes failure detection more accurate. Section 6.6.1 discusses more refined ways of detecting failures.

Subscription Patterns

The operation of compacting a table of level i into a line of the table of level $i + 1$ is called *condensation function* in Capt'n Cook. In our case, this function consists of the three following operations:

Regroup patterns: To represent the interests of all participants of the table, the subscription patterns of the respective participants must be regrouped. This is done in a way which avoids redundancies, i.e., not just by simply forming a conjunction of the individual patterns. A simple example of optimizing accessors representing simple method invocations is depicted in Figure 4.16.

Count participants: The total number of participants at any level can be very useful for several kinds of heuristics. In particular, it enables the estimation of the number of gossip rounds necessary to complete the infection of all concerned participants.

Select delegates: Delegates have to be chosen based on a deterministic characteristic, since all participants in the same subsystem of level i must decide on the same set of delegates without explicit agreement. Currently, delegates with the smallest addresses are chosen. Alternatively, one could take into consideration other criteria associated with participants, like their resources in terms of computing power or memory, or also the nature of their subscription patterns, to reduce the amount of "pure" forwarding of delegates, i.e., handling events as delegate on behalf of other participants, without being itself interested in these events. This optimization task is however not trivial: one can choose participants such that they *individually*, or *altogether*, cover as many interests of the represented participants as possible.

Example Scenario

We depict the view of a small system in the case of multicasting based on type information. Subscriptions are made to unrelated types, e.g., A, B, and more fine-grained subscription patterns are not considered to keep this illustration intuitive.

We map IP addresses straightforwardly to our logical addresses ($l = 4, \forall 0 \leq i \leq l-1\ a_i = 256$). Every participant has a table representing its view of level 1, its view of level 2, a.s.o. Consider a possible configuration of the views for several participants sharing branch address 128.178.73, illustrated in Figure 6.5. The selected redundancy level R is 3. 128.178.73.3 is delegate of level 3, which means that it is known by all participants with $x_3 = 128$.

View of Level 4

x_3	Types of interest	$x_2.x_1.x_0$
3	A, C, D, E	2.230.23, 18.2.78, 188.203.99
18	B, C, E, F	12.2.183, 12.34.24, 180.37.217
128	A, B, C, D, F	3.2.230, 18.120.2, 56.12.234

View of Level 3 ($x_3 = 128$)

x_2	Types of interest	$x_1.x_0$
3	A, B, C	2.230, 18.2, 188.203
18	B, C, D	120.2, 122.34, 180.37
56	C, F	12.234, 18.220, 173.3
178	A, B, C, F	41.21, 73.3, 88.10

View of Level 2 ($x_3.x_2 = 128.178$)

x_1	Types of interest	x_0
41	A, C	21, 23, 24
73	A, B, C, F	3, 17, 19
88	B, F	10, 13, 78
98	A, B, C	15, 17, 128
110	C	1, 6, 7

View of Level 1 ($x_3.x_2.x_1 = 128.178.73$)

x_0	Types of interest
3	A, C
17	A
19	C, F
115	F
116	A, B, F
119	B, C
124	A, B
223	B

Figure 6.5: Hierarchical Membership View

6.5 Hierarchical Probabilistic Multicast (*hpmcast*)

In this section we present *hpmcast*, our gossip-based multicast algorithm which is based on the hierarchical membership outlined in the previous section.

6.5.1 *hpmcast* Algorithm

The algorithm presented for *hpmcast* in Figure 6.6 differs from the *pmcast* algorithm presented in Figure 6.2 mainly by applying the above-mentioned hierarchical multicast scheme, and by furthermore making the inherent performing of garbage collection based on an estimation of the number of necessary rounds more explicit.

Level-Wise Multicasting

As we discussed earlier, the system is pictured as a hierarchy, and the multicasting procedure follows this structure. An event is first propagated in the highest level, from where it moves down level by level. As a consequence, the effective gossips, besides conveying events, also contain the level in which the event is currently being multicast.

Receiving. Upon reception of a gossip, the information about the level is used to place the event in the corresponding gossip buffer. To ensure that the event passes from one level to the next, it is crucial that a participant at level i gossips a received event in any level $i' < i$, and thus also remains in the view of any of these subsequent levels.

Multicasting. A participant would only require gossip buffers from the highest level at which it appears, down to the bottom level, since it will not receive messages from higher levels. However, when HPMCAST-ing, it is reasonable that a participant takes part in the entire gossip procedure at all levels, especially at the topmost one. This increases the probability that an event is well propagated in contrast to a simple scheme where a new event would simply be sent once to a subset of the delegates forming the upmost level. Also, since the membership is dynamic, a participant can be "bumped up" to a higher level at any moment, as well as it can be "dropped" to a lower level.

When a participant HPMCASTs an event, and that event is only of interest for the top-level delegates of a single particular top-level subgroup (e.g., the participant's own subgroup), the first round can easily be skipped, in order to reduce the load on the top-level delegates.

A feigned multicast. Note here that this multicast algorithm does not comply with the notion of genuine multicast proposed in [GS01]: a genuine multicast differs from a *feigned multicast* by the *minimality property*: only participants which are interested in the considered event are effectively involved in the algorithm. Here, a delegate can be involved in the dissemination at a given level, though it is not itself interested. Just like *rfpbcast*, *hpmcast* is a feigned multicast according to [GS01], though one can expect *all* participants to be infected iff $p_1 = 1$. With *rfpbcast*, this is much more likely to occur, since it is the declared goal of that algorithm to treat

```
Executed by participant_i

initialization
view[1..l]
gossips[1..l] ← ∅

task GOSSIP                                                   {every P milliseconds}
  for all level ∈ [l..1] do
    for all (event, prob, round) ∈ gossips[level] do
      if round < ROUNDS(level, prob) then                    {not too many round}
        round ← round + 1
        dests[1..F] ← ∅
        repeat F times                                       {choose potential destinations}
          dest ← RANDOM(view[level] - dests)
          dests ← dests ∪ {dest}
          if event ◁ dest then
            SEND(event, prob, round, level) to dest
      else
        gossips[level] ← gossips[level] \ {(event, prob, round)}
        if level > 1 then
          gossips[level-1] ← gossips[level-1] ∪ {(event, GETPROB(level-1, event), 0)}

upon RECEIVE(event, prob, round, level): do
  if ∄ level ∈ [1..l] ∃ (event, ..., ...) ∈ gossips[level] then   {buffer and deliver}
    gossips[level] ← gossips[level] ∪ {(event, prob, round)}
    if event ◁ participant_i then
      HPDELIVER(event)

upon HPMCAST(event): do
  gossips[l] ← gossips[l] ∪ {(event, GETPROB(l, event), 0)}

function ROUNDS(level, prob)                                  {expected number of rounds}
  return $\log(|view[level]|\ R\ prob)\ (\frac{1}{\log(F\text{prob}+1)} + \frac{1}{(F\text{prob})})$

function RANDOM(from)                                         {choose random participants}
  return dest ∈ from

function GETPROB(level, event)             {probability of matching this event at this level}
  hits ← 0
  for all dest ∈ view[level] do
    if event ◁ dest then
      hits ← hits + 1
  return $\frac{\text{hits}}{|\text{view[level]}|\ R}$
```

Figure 6.6: Hierarchical Probabilistic Multicast Algorithm

participants regardless of their interests when disseminating, and to filter events only locally before possibly passing them to the application (layer).

Parameters

As previously outlined, the expected number of rounds can be used to estimate the number of rounds necessary to disseminate a given event. Note here that this does not require participants to be synchronized, nor does it make any assumption on delivery delays. The computation of this expected number of rounds relies however on several parameters, like the fraction of interested participants, or the average message transmission loss (Equation 6.9).

Fraction of interested participants. The probability p_1 that a given participant is interested in a particular event at a given level can be simply measured by matching that given event against all the subscription patterns of all participants for that given level. This is a costly operation, but is only performed by a maximum of R participants at each level except the topmost one, since this is the maximum number of processes infected initially in a subsystem. At the upmost level, only the participant effectively publishing the event will perform this matching.

Since delegates represent several participants, the measured probability can be expected to be higher than p_1 (except for the lowest level), according to 6.14.

Expected number of rounds. With the fraction of interest, the expected number of rounds for a given level can be computed. This requires the number of delegates forming the level (with respect to the subsystem of the considered participant), which is given by multiplying the number of different subsystems of level $i-1$ in the view of level i, i.e., the number of lines in the corresponding view table (noted $|view[i]|$ in Figure 6.6), by the number of delegates for each subsystem. If this number of delegates is not a system constant, one can alternatively simply use the total number of delegates in the view of level i.

Environmental parameters. Environmental parameters, such as the probability of message loss, or the probability of a crash failure of a participant, are to be considered when computing the expected number of rounds necessary to spread an event. These are however more difficult to approximate ([EKG01]), especially the latter one. Like in most gossip-based algorithms, where simulations or analytical expressions enable the computing of "reasonable" values for parameters such as hops or forwards, choosing conservative values is the best way of ensuring a good performance. (For simplicity, these parameters have not been added in the algorithm.)

6.5.2 Analysis

For analysis, we presuppose a "regular" hierarchy, i.e., for any participant, all branch addresses derived from that participant's address $x = x(l-1).\cdots.x(0)$ have the same number of subsystems, which we denote by a.

$$\begin{gathered}\forall x, i \ x = x(l-1).\cdots.x(0), 1 \le i \le l \\ |x(l-1).\cdots.x(i)| = a \ \le \ a_i\end{gathered} \tag{6.15}$$

Accordingly, the total number of participants in the system is simply given by

$$n = a^l \tag{6.16}$$

Also, we consider that, with respect to a given event, the participants interested in that event are uniformly distributed over the entire system.

View Size

Every participant must know the delegates of every level as shown in Figure 6.4 (for $l = 4$ and $R = 3$).

According to Equation 6.12, a participant knows the following number of participants for the different levels of a regular hierarchy:

$$m_i = \begin{cases} R\ a & 1 < \ i \ \le \ l \\ a & \qquad i \ = \ 1 \end{cases} \tag{6.17}$$

which adds up to a total of

$$\begin{aligned} m \ &= \ \sum_{i=1}^{l} m_i = R\ a\ (l-1) + a \\ &\in \ O(l\ R\ n^{1/l}) \qquad l \ge 2 \end{aligned} \tag{6.18}$$

Expected Number of Rounds

Based on Equation 6.14, we can determine that, in a regular hierarchy,

$$p_i = 1 - (1 - p_1)^{a^{i-1}} \tag{6.19}$$

Furthermore, the number of expected rounds can be approximated by the sum of the rounds spent at each level of the hierarchy:

$$T_{tot} = \sum_{i=1}^{l} T_i = \sum_{i=1}^{l} T(m_i\ p_i, F\ p_i) \tag{6.20}$$

This expression is pessimistic, by neglecting the very fact that every interested subsystem, except the topmost one, starts with an expected number of infected participants which is bigger than 1, namely R.[10] These delegates already being infected, we can obtain a more precise expression by subtracting at each level the time necessary to get from 1 to R infected participants:

$$T'_{tot} = \sum_{i=1}^{l} T(m_i\ p_i, F\ p_i) - (l-1)\ T(R, F) \tag{6.21}$$

The probability in the terms added in Equation 6.21 reflects that, at each level, every interested subsystem is represented by R delegates, which are all (probability of 1) interested in the considered event.

Number of Infected Participants

A precise analysis of the spreading of the event in time, yielding a distribution of the probability for the infection of a fraction of the system as in the above algorithms, introduces a high complexity in this case due to the decomposition of the entire system into subsystems. (At the lowest level, the number of different states is potentially $a^{(a^{l-1})}$ for a given number of rounds.) One can however reuse the same Markov chain introduced previously to compute the expected number of infected participants (or an approximation as in [Bai75], or [EKG01]) in a subsystem of level i $(1 \leq i \leq l)$ after gossiping at that given level:

$$E[s_{T_i}] = \sum_{j=0}^{m_i\ p_i} P(m_i\ p_i, F\ p_i)[s_{T_i} = j]j \tag{6.22}$$

This is again a pessimistic value, since we do not consider the possibility that the subsystem $(1 \leq i < l)$ initially comprised more than one infected participant (cf. 6.21).

We are now able to compute the probability that an "entity" of level i is infected after gossiping at that level (provided the corresponding subsystem of level $i+1$ was initially infected):

[10]To be fully accurate, we would also have to consider that the number of participants to infect at the topmost level is in the general case given by $m_l p_l + 1$, since the HPMCAST-ing participant is the initially infected one. That participant would have to be furthermore considered in any of its own subsystems.

$$r_i \; = \; 1 - \left(1 - \frac{E[s_{T_i}]}{m_i \; p_i}\right)^{\frac{m_i}{a}} \tag{6.23}$$

For all levels, except the lowest one, an "entity" of level i means a subsystem of level i (that is, its R delegates for that level). At the lowest level, an "entity" refers to a participant. $r_i|_{i=1}$ hence simply resumes to $\frac{E[s_{T_i}]}{a \; p_i}$, the expected fraction of participants infected when gossiping in a system of level l.

Provided that $g_{i+1} = j \leq a^{(l-i)}p_{i+1}$ entities were infected at level $i+1$ (in total), the probability of ending up with $g_i = k$ entities infected at level i $(1 \leq i \leq l)$ is given as follows:

$$\begin{aligned} p_{ijk} &= P[g_i = k | g_{i+1} = j] \\ &= \begin{cases} P[g_{i+1} = j]\binom{j \; a \; p_i}{k} r_i^k (1-r_i)^{j \; a \; p_i - k} & k \leq j \; a \; p_i \\ 0 & k > j \; a \; p_i \end{cases} \end{aligned} \tag{6.24}$$

Finally, we can compute the probability of having k entities infected at level i:

$$P[g_i = k] \; = \; \begin{cases} 1 & i = l+1, k = 1 \\ 0 & i = l+1, k \neq 1 \\ \sum_{j=k/(a \; p_i)}^{a^{(l-i)}p_{i+1}} P[g_{i+1} = j] p_{ijk} & 1 \leq i \leq l \end{cases} \tag{6.25}$$

Expected Number of Infected Participants

Based on the upper equation, we are able to express the expected number of infected entities g_i after gossiping in a subsystem at level i $(\leq i \leq l)$

$$E[g_i] \; = \; r_i \; a \; p_i \tag{6.26}$$

Consequently, the expected number of totally infected participants is given by the following expression:

$$\prod_{i=1}^{l} E[g_i] \tag{6.27}$$

The expected reliability degree can be simply obtained by dividing the upper expression by the number of effectively interested participants, $n \; p_1$.

6.5.3 Simulation Results

We have simulated *hpmcast*, by simulating successive rounds, and observing the spread of a single obvent. Different parameters of the algorithm have been varied. We use these results to pinpoint the limitations of the basic variant of *hpmcast* presented throughout this section. Ways of counteracting these limitations, not considered here for simplicity of presentation, will be discussed in Section 6.5.4.

Interest

On the one hand, Pittel's formula gives extremely good results when the system grows in size. This is visible in Figure 6.7, which shows a very good overall degree of reliability. On the other hand, smaller systems, and hence "rather small" values for p_1, are not well captured by this asymptote. Figure 6.8 zooms in on the previous figure, to show the decrease of reliability with small values for p_1. Indeed, the computed expected number of rounds increases first with a decreasing p_1, before decreasing quickly and becoming 0 in $n\ p_1 = 1$ (Figure 6.9). Though this last value reflects reality, the asymptote gives less accurate information towards that value. Even with a better approximation this problem can be observed, since the stochastic approaches underlying epidemiology, and hence gossip-based algorithms, indeed reflect the interest of large populations. This loss in reliability was hence expected, and there are several ways of counteracting it (see Section 6.5.4).

More precisely, the interpretation of "rather small" depends on the considered level, and hence on p_i, but also on m_i. In fact, p_i becomes bigger at every level i, and hence, the number of potentially interested participants increases, making higher levels less prone to this type of undesired effect. Lower levels, especially the lowest one, are most likely to reach critically low sizes, since their p_i become smaller. At the level $i = 1$, m_i is furthermore smaller than in all above layers.

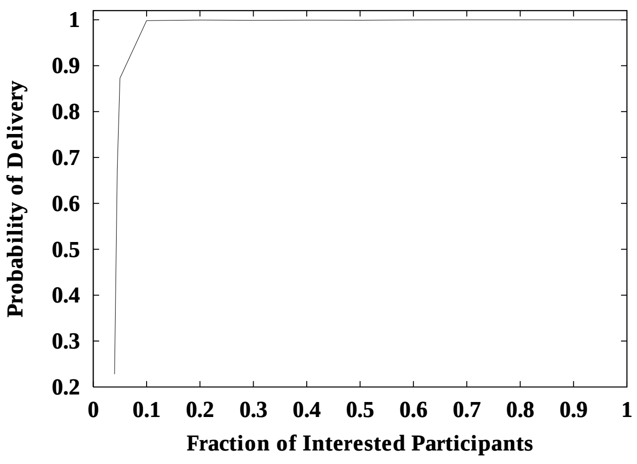


Figure 6.7: Varying p_1; $n \approx 10000$ $(a = 22), l = 3, R = 3, F = 2$

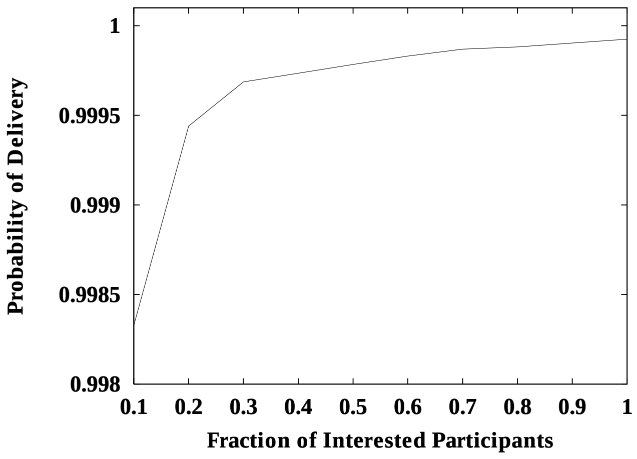


Figure 6.8: Reliability Depending on p_1; $n \approx 10000$ $(a = 22), l = 3, R = 3, F = 2$

Fanout

As typical for gossip-based algorithms, increasing the fanout decreases the number of rounds necessary to infect the concerned participants. Since in our case we have not considered the gossiping of digests for mutual updates based on gossiper-pull, but limit the entire propagation of events by the number of expected rounds, the estimation of this latter value has a certain impact on reliability.

Scalability

It is however not clear whether, and how, the performance of *hpmcast* is impacted when increasing the scale of the system. For the above reasons, a slightly increased number of participants leads to a better approximation of the number of necessary rounds, and can even lead to a better reliability degree. On the other hand, one can also expect that increasing the size of the system, and hence the number of expected rounds, also increases the potential difference between the latter value and the *effective number* of necessary rounds.

As conveyed by Figure 6.11 in any case, *hpmcast* shows very good scalability properties when increasing a in a hierarchy of fixed depth (the system size increases following $a^l!$). The scalability however depends on the proportion of interested participants. With a small p_1, the above-mentioned problem manifests itself in a slightly stronger decreasing reliability with an increased system size. This effect is however counteracted by the better approximation of the number of necessary rounds with larger (sub)system sizes.

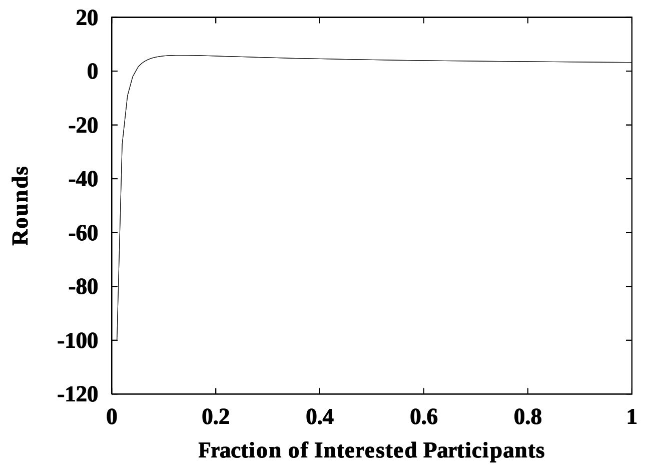


Figure 6.9: Rounds Depending on p_1; $n = 100, F = 2$

Hierarchy Depth

Similarly, one can expect that increasing the hierarchy depth has a similar impact on the reliability degree as decreasing p_1, since one can expect the size m_i of the individual groups to decrease, and reach quickly a critically small size.

This can be indeed observed in certain cases, just like the opposite. The various intervening parameters seem to hinder the appearance of any clear trend.

Redundancy

Increasing redundancy obviously increases reliability, since the probability that at any given level an interested subsystem becomes isolated decreases by increasing R. This redundancy however has less impact on the main problem encountered with small p_1, since this problem appears first at the lowest level, and any redundancy factor $R > 1$ leads to multiplying the number of potentially interested participants by R. Hence, when further increasing R, only little more reliability is gained (see Figure 6.12).

6.5.4 Tuning *hpmcast*

We have considered several possibilities of tuning *hpmcast* with respect to small values for p_1.

Possibilities

Besides adding a gossip-based exchange phase for digests of received events (periodically sending identifiers of buffered events to interested participants), one can

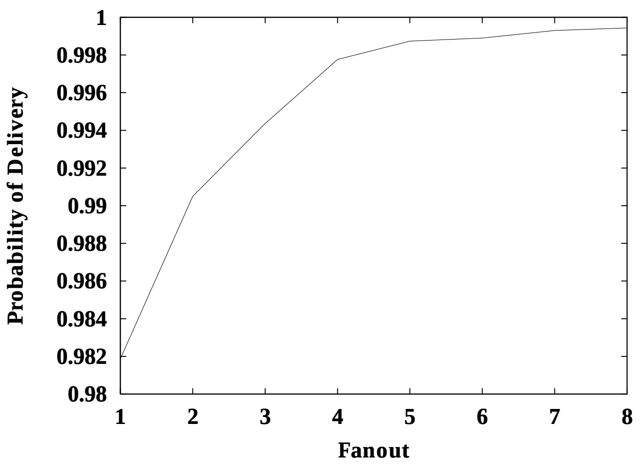


Figure 6.10: Reliability Depending on F; $n \approx 10000$ $(a = 22), l = 3, R = 3, p_1 = 0.2$

basically distinguish two ways of improving the performance of *hpmcast*. These are namely, (1) using Pittel's asymptote, however increasing artificially the number of interested participants, and (2) by applying another approximation of the number of necessary rounds. The second approach can for instance be achieved by using a more rough approximation of $T(n, F)$, possibly associated with an expected reliability degree (e.g., [Bai75]).[11] We have however been more attracted by the accuracy offered by Pittel's asymptote, and are more interested in very large systems (even exceeding the feasibility of a simulation), where an absolutely small p_1 still leads to a large number of interested participants in a lowest-level group.

Increasing the Audience

We have adopted a more pragmatic approach, consisting in increasing the audience by adding participants to the set of interested participants. To that end, we have modified the above algorithm to include non-interested participants if the number of interested participants in the system drops below a threshold D. In that case, every involved participant decides that the D first participants in the view of the corresponding level are interested, in addition to the remaining effectively interested participants. By fixing a lower bound on the desired reliability degree, D can be obtained through analysis or simulation. The result of such a tuning is illustrated by Figure 6.13, which compares the original degree of reliability with the one obtained after tuning *hpmcast*.

[11]With such an approach, the term $\frac{E[s_{T_i}]}{m_i\ p_i}$ in 6.23 can be directly replaced by the expected reliability degree of gossiping in a system of size $m_i\ p_i$.

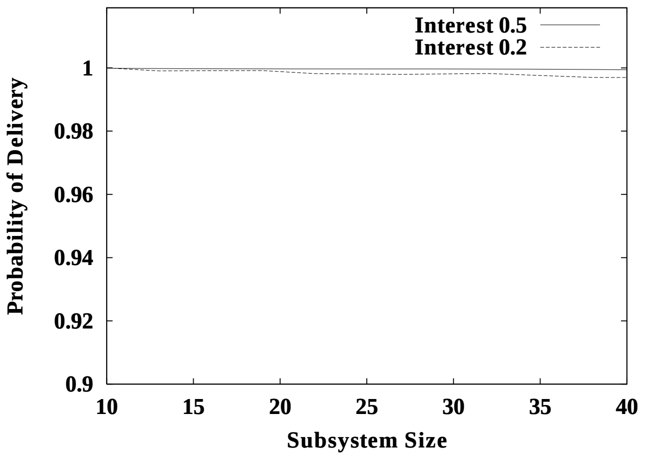


Figure 6.11: Scalability when increasing a; $l = 3, R = 4, F = 3$

6.6 Discussion

We discuss several issues related to our hierarchical approach outlined above, including the effect of increasing the depth of the hierarchy.

6.6.1 Exploiting Hierarchies

The hierarchical organization of participants can be exploited for several aspects of reliable event delivery.

Filtering

Higher-level delegates receive more gossips than lower-level participants. One way of reducing the load on these delegates could consist in applying filters with a weaker accuracy, in order to speedup filtering.

Filter coverage. More precisely, when filtering at a high level, events might successfully pass this filtering, with respect to a set of participants, though these events are of interest for none of these participants, nor for any of the participants they might represent. This idea is based on the notion of *filter coverage* [M0̈1], where a first filter is said to be *covered* by a second filter if everything that passes the first filter also passes the second one (but not necessarily vice-versa). By applying such a weaker filter instead of the covered precise filter, filtering cost can be substantially reduced.

Event coverage. The difference becomes more important if the format of the filtered events follows this modified accuracy. For instance, (parts of) events can

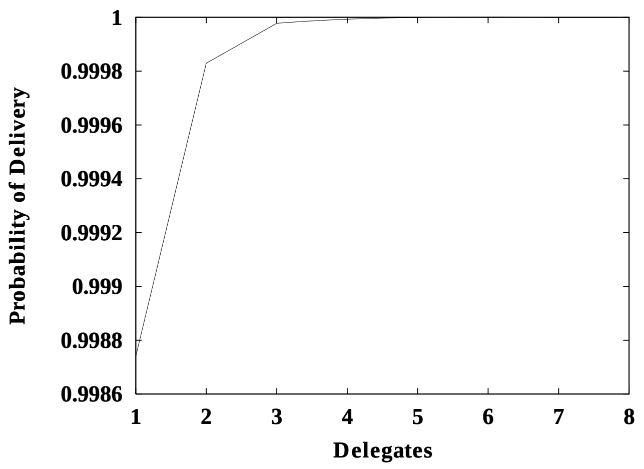


Figure 6.12: Reliability Depending on R; $n \approx 10000$ $(a = 22), l = 3, F = 5, p_1 = 0.2$

be reflected by more primitive representations, such as XML structures transferred with the effective event objects. This can avoid the deserialization of events every time they are filtered. At a very first level, an event can for instance even be viewed as "data plus a type identifier".

Example. A resulting mapping of filter and event coverage to hierarchy levels could be the following (by increasing hierarchy levels):

Method invocations: The original filter expressed on arbitrary method invocations, as in the model presented in Section 3.5, is applied in the subscriber's process, to ensure that the subscriber receives *exactly* what was specified through the filter.

Attribute comparisons: The next level already works with an XML-like representation of events, reflecting their attributes. Attribute comparisons, though expressed through access methods in the initial filter code, are mapped to reads of the corresponding entries of the XML structure with comparisons. Only such simple comparisons are applied at this level, which presupposes that the middleware has the possibility of identifying the attributes of arbitrary event objects.[12] This can be for instance achieved through conventions on the names of corresponding methods (`getxxx()`). Filters applied at this level represent a simple conjunction (without redundancies) of the filters regrouped from the different subscribers at the next hierarchy level.

Selective comparisons: At the next level, only more certain attributes are verified. For instance, attributes for which a subsystem gives different ranges of values of

[12]Note here that, on the one hand, directly accessing attributes in *filters*, in a way controlled by the middleware, can lead to a considerable benefit in terms of performance, and should hence not be precluded. Supporting attributes as the *only* properties of events for describing *subscription patterns*, on the other hand, is very compelling and should be prohibited.

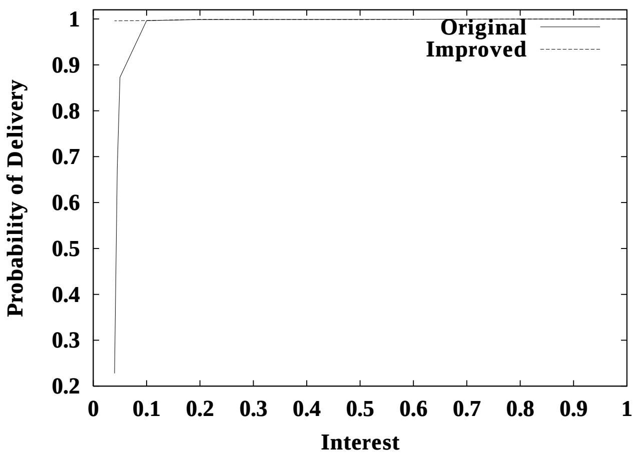


Figure 6.13: Varying p_1; $n \approx 10000$ $(a = 22), l = 3, R = 3, F = 2$ with tuned *hpmcast*

interest are not filtered anymore. Also, filters for related types, which are in a subtype relation, are replaced by a compound filter possibly only checking events for conformance with the most general type. In a hierarchy with more levels, one could imagine several degrees for this selectivity.

Type conformance: At the highest level, events are only filtered based on their event type. As suggested above, type information can even be attached as an identifier encoded in a set of bytes. This circumvents any deserialization of events, or parsing of XML descriptions.

When covering filters, the effective probability p_1 of interest for any given participant cannot be as easily computed based on the previous level. In contrast, without weakening filters, one could compute an estimation of the probability p_i at a given level based on p_{i+1} and Equation 6.19, and then improve that approximation by successively matching the event against F participants and adjusting the effective matching rate.

Dissemination

The hierarchical disposition of participants can also be used to apply different dissemination media. As an example, a UDP Broadcast could be used at the lowest level instead of gossips, if for instance the lowest level is characterized by highly populated LANs.

One can also imagine using a more reliable network protocol, also for intermediate levels. Depending on the desired compromise between reliability and performance, a “more” reliable algorithm can be very advantageous at the upmost level. Indeed, an unsuccessfully completed attempt of infecting a higher-level “entity” induces a domino effect, by isolating the entire subsystem.

Failure Detection

Last but not least, the membership can be implemented in a way that applies different failure detection mechanisms at different levels. For instance, delegates of a given level can decide to autonomously exclude other participants if they have not received any gossips from them for some time.

At a lower level, e.g., LAN, participants can apply a less "passive" style, by actively pinging participants from which no gossips have been received for a long time before effectively removing them from views.

6.6.2 Broadcasting with *hpmcast*

It might be interesting to apply our hierarchical membership to the broadcasting of gossips. We show here that the number of expected rounds necessary to infect the entire system does not depend on the use of a hierarchy, more precisely, on the number of levels in the hierarchy, in the case of a broadcast ($p_1 = 1$).

In fact, for $p_1 = 1$, the number of rounds necessary to multicast an event is similar to the number of rounds in the non-hierarchical algorithm (*pmcast*), or, in other terms, $T_{tot} = T(n, F)$.

Indeed, according to Equation 6.21 (and 6.17):

$$\begin{aligned}
T'_{tot} &= (l-1)\ T(a\ R, F) + T(a, F) - (l-1)\ T(R, F) \\
&= \left[log\ (R\ a)^{l-1} + log\ a - log\ (R^{l-1})\right] \left(\frac{1}{F\ (1-\epsilon)(1-\tau)} + ... \right) \\
&= (log\ n) \left(\frac{1}{F\ (1-\epsilon)(1-\tau)} + \frac{1}{log(F\ (1-\epsilon)(1-\tau)+1)} \right) \\
&= T(n, F)
\end{aligned} \tag{6.28}$$

Consequently, the depth of the hierarchy has no impact on the time it takes to broadcast an event in the entire system.

This might seem surprising at first glance, since in this *hpmcast* algorithm, an event is *simultaneously* gossiped in a^{l-i} distinct subsystems at level i. As however already pointed out in [Pit87], a gossiped event spreads slowly at the beginning, since very little participants are infected and can hence infect susceptible participants, as well as towards the very end, since an increasing fraction of the system is already infected, and these "absorb" many gossips. In between, the spreading proceeds quickly. Increasing the size of the system only increases the number of necessary rounds logarithmically (6.4), which makes gossip-based approaches so attractive for large scale settings.

6.6.3 Levels

We analyze the effect of increasing the depth of the hierarchy on the performance of the system, through several aspects.

View Size

The first considered measure is the size of the view that every participant has for its corresponding subsystem.

We can show that the number of participants that a given participant knows is minimal with a hierarchy level $l = log\ n$. More precisely, m decreases as l increases, as long as $l\ < log\ n$.

Indeed, based on 6.18, and the fact that

$$\frac{d(n^{1/l})}{dl} = -\frac{log\ n}{l^2}\,n^{1/l} \tag{6.29}$$

we can see that m decreases as long as $l \leq log\ n$:

$$\begin{aligned} \frac{dm}{dl} &= R\ n^{1/l}\left(1 - \frac{log\ n}{l}\right) \\ &<\ 0 \qquad \forall l < log\ n \end{aligned} \tag{6.30}$$

In fact, $l = log\ n$ is a minimum:

$$\begin{aligned} \frac{d^2m}{dl^2} &= \frac{R\ n^{1/l}\ (log\ n)^2}{l^3} \\ \left(\frac{d^2m}{dl^2}\right)_{l=logn} &= \frac{R\ e}{log\ n} \\ &> 0 \end{aligned} \tag{6.31}$$

In practice, this limit is probably never reached, since in most cases R will be chosen such that $R > e$:

$$\begin{aligned} & R > e \\ \Rightarrow & a > e && (a \geq R) \\ \Rightarrow & l < log\ n && (a = n^{1/l}) \end{aligned} \tag{6.32}$$

Failure Sensitivity

Increasing the depth of the hierarchy however also has effects on the stability of the membership.

Participants to be updated. Increasing R improves fault tolerance, by reducing the probability of partitioning, and the probability of isolation of an interested subsystem. This is however not the only measure of reliability. For instance, the average number of participants whose membership views have to be updated upon failure of any given participant decreases as l increases.

For a given level i, we have d_i participants:

$$d_i = \begin{cases} R\ a & \qquad i = l \\ R\ a^{l-i}\ (a-1) & 1 < \ i < l \\ a^{l-1}\ (a-R) & \qquad i = 1 \end{cases} \tag{6.33}$$

The number of participants which know a given participant of level i, i.e., the number of participants which have to be updated upon failure of a given participant of level i is

$$u_i = a^i - 1 \quad 1 \leq i \leq l \tag{6.34}$$

The average number of participants which have to be updated after a failure is hence given by

$$\begin{aligned} u_{upd} &= \frac{1}{n}\sum_{i=1}^{l} d_i\ u_i \\ &= (a-1)[R\ (l-1)+1] \end{aligned} \tag{6.35}$$

which in fact corresponds to the number of participants known by every participant *without considering duplicates*, i.e., by counting a delegate at level i only for that level, and not for lower levels. This is not really surprising, since every participant knows u_{upd} distinct participants, and hence, in average, a participant is known by u_{upd} participants. $u_{upd} \in O(ln^{1/l})$ scales well with n, and decreases similarly to m.

The failure of a participant which is not delegate can be dealt with more easily than the failure of a delegate. The latter failure type involves more membership updates. By increasing the depth l of the hierarchy, the number of participants which are delegates of some level increases. However, one can expect the number of participants that have to be updated to decrease: a hierarchy of 1 level will see the updating of all participants in the system upon the failure of a single participant, while in a hierarchy of 2 levels already, a failed participant is with a big probability only known by approximately a number of participants equal to the square root of the size of the system, while only R times that same number of participants are known by all participants, etc.

Average level of a participant. This is however not the only measure of stability for the hierarchy. Indeed, if a participant of a higher level fails, the information about its failure will have to travel over more levels, which becomes increasingly important if such updates are piggybacked by event gossips, like in our case.

In turns out that the average number of levels of the hierarchy that an update will travel increases with the depth of the hierarchy, since the average level of a participant in the hierarchy increases as the number of levels increases.

This result can be intuitively easily explained, since in a hierarchy consisting of only 1 level, any participant can potentially only reach level 1, while in a hierarchy of level 2 already, several participants will reach level 2, etc.

Consider the average level l_{upd} of all participants given by the following expression:

$$\begin{aligned}
l_{upd} &= \frac{1}{n}\sum_{i=1}^{l} d_i \; i \\
&\approx \frac{1}{n}\sum_{i=1}^{l} R \; (a-1) \; a^{l-i} \; i \qquad (large \; l) \\
&\approx R \; a \int_{i=1}^{l} a^{-i} \; i \; di \\
&= \frac{R \; a}{log \; a}\left(i \; a^{-i} - \frac{a^{-i}}{log \; a}\right)_{i=1}^{l} \\
&= \frac{l^2 \; R \; n^{1/l}}{n \; log \; n}\left(1 - \frac{1}{log \; n}\right) + \frac{l \; R}{log \; n}\left(\frac{l}{log \; n} - 1\right)
\end{aligned} \tag{6.36}$$

Both of these remaining terms increase as l increases.

Hence, increasing the number of levels in the hierarchy, as intuition suggests, cannot be done indefinitely without side effects. Indeed, as already illustrated by simulation results, similar tradeoffs can be expected concerning the dissemination of the events in the hierarchy. Parameters, such as the number of total rounds, the obtained reliability degree, but moreover also the number of infected participants which are not themselves interested in a given event, but only act as delegates, and the filtering load on an average participant might probably not be modified without affecting the others.

Summary

The Hierarchical Probabilistic Multicast (*hpmcast*) algorithm presented in this chapter is a novel gossip-based algorithm, which abides well to strongly dynamic and largely scaled settings, such as TPS. The principles adopted in *hpmcast* are nevertheless general, and do not only make sense in our precise context of TPS, but in any form of completely decentralized, peer-to-peer architecture.

hpmcast exploits locality and redundancy properties of the underlying system, and intrinsically reduces the view of each participant.

The presented analysis and simulations convey the strong scalability of *hpmcast*, and enable the trimming of parameters, e.g., based on a prediction of the upper bound on the number of participants in the system. As a result of the inherent compromises between certain faces of scalability, e.g., memory, as well as computing and network resources, it is difficult to define optimal values for parameters such as the depth of the hierarchy.

Chapter 7

Conclusions

This thesis (1) introduces TPS, a higher-level variant of the publish/subscribe paradigm, (2) expresses it in Java through both a library and a language integration approach, and (3) introduces an original and strongly scalable multicast algorithm to implement it.

Before debating strategic directions for future work, we summarize our experiences gathered with TPS by opposing our library and language approaches, in an attempt of subsequently drawing conclusions on language mechanisms in favor of "clean" implementations of libraries for TPS, with some side effect conclusions on distributed programming in general [DEG00].

Library vs Language: The Case of TPS

A question that naturally arises is, whether (in general) a library or a language integration approach to TPS is a better fit. We attempt to answer this question in the case of Java by comparing our two approaches, emphasizing a concise set of different criteria (summarized in Figure 7.1).

	Library	Language Integration
Type Safety	−	+
Simplicity	−	+
Readability	−	+
Flexibility	+	−
Performance	=	=

Table 7.1: Library vs Language Integration: Summary

Type Safety

By all evidence, the language extension provides for more type safety than the library approach. Let aside the fact that to obtain type safety in obvent delivery (Section 3.6.1) with the library, we had to make use of genericity currently not part of Java, type safety in obvent dissemination (with respect to subscription patterns) could only be achieved unsatisfactorily. Indeed, filters, though in principle based on the invocation semantics of Java, can only be type-checked at runtime.

We do not take into account the fact that these filters permit the expression of a primitive form of structural conformance, since this rather represents a side effect unrelated to the original motivations for TPS, and can with some effort also be achieved in the language integration approach.

Simplicity

Simplicity represents a scale for characterizing *how easily TPS can be put to work by a developer.* One can obviously always add more features to a language in order to allow for very concise TPS statements, which however might as well involve a more important learning phase. A library API, on the other hand, can be clean and simple to use for developers aware of similar systems, but can as well turn out be very elusive.

In the present case, programming with a library involves learning a rather complex API for describing subscription patterns, while Java$_{PS}$ uses the native Java syntax. The advantage of the latter solution is however diminished, since only very limited semantics are associated with subscription patterns for the sake of making them easily transferable and optimizable. Unexperienced developers will be strongly tempted to trespass these restricted semantics, while the filter library used in conjunction with GDACs inherently enforces its boundaries.

In short, both approaches have their shortcomings with respect to simplicity, though an advantage must be awarded Java$_{PS}$.

Readability

Readability is a measure of *how easily a developer can understand TPS-related code* developed by a third party. Concrete issues underlying readability include, how concise and clear TPS-related code is, and where the code is actually located.

Concerning this issue, the language integration approach clearly overtrumps its rival. Subscription expressions are very concise, and nevertheless expressive and clear, by promoting native Java syntax.

Subscribers in the library approach are implemented in specific classes. Though in Java these can be "inlined" (by defining anonymous classes on the fly), their syntax is less concise than the TPS-specific subscription expressions in Java$_{PS}$, and a subscription in the library approach involves a GDAC, which has to be set up previously. More importantly however, subscription patterns in the library approach are only poorly readable.

Flexibility

With the *flexibility* of an implementation of TPS we mean *how easily developers can apply TPS to specific domains.* This aspect is important, because an implementation of TPS, which only supports a limited problem domain, can only be of modest interest in the face of a language integration.

As one might expect, the library approach offers more flexibility than the language integration approach, by presenting several ways of performing customizations. For instance, a developer can decide to define very specific filters (conditions) [EG01a], and skilled developers can decide to devise their own GDACs, leading to new QoS (though this increased flexibility nourishes the potential mismatches in QoS).

In the case of the language integration approach, new conditions can be encapsulated inside individual obvent types, however only prior to deployment, and QoS are concealed behind predefined obvent types. Though new types can easily added by the engine develops, it becomes more difficult for the application developer to "plug in" own algorithms.

Performance

In terms of performance, we have observed similar behavior in both cases, provided that both approaches made use of the same subscription patterns, since the same model underlies subscription patterns in both approaches, and these patterns are converted to similar representations for the sake of optimizations.

Language Mechanisms for TPS

As a logic continuation, we tackle here the issue of identifying concepts that a language would have to incorporate to enable a TPS library implementation satisfying all our requirements outlined in Section 3.1.3. Ultimately, the answer to this question might also give useful information for other distributed programming paradigms.

Clearly, obvent types could be precompiled to obtain something similar to the typed adapters outlined in the language integration approach. We are here however thinking of a language incorporating mechanisms which would make any specific compilation obsolete.

Type Safety

Part of the question is thereby already answered. Subscriptions are inherently parameterized by the considered obvent type, i.e., a subscription pattern and its associated handler deal with the same, arbitrary type of obvents. To ensure type safety, yet avoiding any specifically generated typed adapters, or automatically inserted type casts, libraries reflecting the association of handlers and filters TPS mandate a form of *genericity.*

Furthermore, since we are in a distributed context, there must be a way for distributed participants making use of the same obvent types to "connect". This requires *runtime type information*, e.g., to reify types. Indeed, verifying how types are related, and performing dynamic type inclusion checks on objects are a sound base for avoiding type errors at obvent dissemination, i.e., the communication infrastructure level. Such reflective mechanisms are commonly viewed as part of *introspection*, or more generally, *structural reflection.*[1]

As shown by our generic library approach, *runtime support for genericity* can be useful to make the "connection" described above safer, and also to support dynamic type checks based on structural reflection, if the filters cannot be checked at compilation.

Subscription Patterns

In Java$_{PS}$, filters are implemented through some form of *deferred code evaluation* to ensure that these can be type-checked at compilation, by nevertheless offering an insight to the middleware. Also, *behavioral reflection*, possibly combined with *operator overloading*, could provide for an ideal compromise of static type-checking and deferred evaluation. A *program reification* in the form of a *parse tree*, such as in Smalltalk (see Section 5.4.4), could address the same requirement.

The requirements posed by filters are the most tedious to fulfill, and there are currently probably only very few statically typed languages which provide sufficient mechanisms.

Subscribers

Subscribers can be represented by methods implemented by callback objects of a specific type, or *closures.* While closures enable the concentration of all subscription-related code, references to such closures (or *higher order functions*) enable the placing of the obvent handler at any point, and methods force the definition of a specific class. With respect to type safety, as mentioned above, the main goal consists in verifying that the signature of the provided piece of code coincides with the filter, i.e., the handler and the filter each have a single formal argument with a coinciding type. In the case of a callback object, to avoid generating specific callback types for each obvent type, *genericity* can be used again to type parameterize the callback type.

[1]In the case of a dynamically typed language such as Smalltalk, this would become increasingly important through the absence of type checks at compilation, as supported for instance by genericity.

Obvents

Subtyping, a key paradigm in object-oriented programming provides the necessary foundation for the ability of updating applications. Especially in a distributed context, the full exploitation of this paradigm is only given by *dynamic class loading*, that is, the possibility of updating existing components by adding new subtypes at runtime.

Also, we have made use of *multiple subtyping* in both the library and language integration approaches to form new QoS expressions from several more basic ones. Though it is not impossible to achieve similar behavior without multiple subtyping as part of the type system, for instance by applying a corresponding design pattern [GHJV95], we believe that any form of expressing multiple inclusion eases QoS expression.

No Limits

The prominent mechanisms stated above have all been investigated in the context of this thesis, and by no means the intention here is to claim that the outlined mechanisms cover all possibilities. As conveyed by the advantages of our language integration approach over the library approach however, Java is not sufficient for TPS.

Establishing a more exhaustive list of languages and features would require a detailed study of all established and prototypical object-oriented languages in all their variants and versions, as well as all existing concepts (including the above-mentioned).

Consider for instance the `myType` type qualifier introduced by Bruce et al. in PolyTOIL, and inherited by its follow-up Loom [BPF97]. In any given method body, this type 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, it 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 obvent classes (without first-class adapters), which subtype a specific root obvent type `Obvent`. Following the Java syntax, one could imagine having something like the following:[2]

```
public class Obvent {
  public myType subscribe(Subscriber<myType> s) {
    ...
    /* return a proxy */
  }
}
public class MyObvent extends Obvent {...}
```

[2]`myType` has a companion, written `@myType`, which denotes the *exact* dynamic type of `this`, i.e., subtypes are considered harmful. To be fully precise, the following example should use `@myType` as type parameter for the subscriber.

Subscribing to an application-defined obvent class, such as the `MyObvent` class above, can be done simply by first creating an instance of that class, and then invoking the `subscribe()` method:

```
class MyObventSubscriber implements Subscriber<MyObvent> {...}
MyObvent proxy = new MyObvent().subscribe(new MyObventSubscriber());
proxy.equals(new MyObvent(...));
```

The impact of every detail of a considered language's type system becomes here visible again. Indeed, the above subscription scheme could fulfill all our requirements in a language which, unlike Java, does not provide any purely abstract types (since the above design does not support subscriptions to abstract types). Furthermore, if the `myType` type qualifier is available in class methods, one could omit creating an instance of an obvent class just for subscribing to that type.

Note that it is more difficult to express subscriptions by subscribing to obvent classes with F-bounded parametric polymorphism. Consider the following example:

```
public class Obvent<myType> {
  public myType subscribe(Subscriber<myType> s) {...}
}
public class MyObvent extends Obvent<MyObvent>{...}
```

Here, one can indeed subscribe to the `myObvent` class as in the previous example. However, the explicitly introduced type parameter `myType` is bound, and hence remains the same in every subclass of `MyObvent`, jeopardizing type safety. Furthermore, compilation cannot ensure that every obvent type is parameterized by itself, the way the `MyObvent` is. On the other hand, the `subscribe()` method can be easily defined as a class method.

Similarly to the `myType` type qualifier, the concept of mixins could enable the merging of the abstraction for subscribing with the very event types; here by "adding" methods expressing subscriptions and unsubscriptions to application-defined event types at runtime, instead of inheriting them from an abstract event type.

Future Work

We are currently pursuing, or foreseeing the orientation of future efforts in several, partly opposed directions.

Subscription Models

As already discussed in Section 3.7.2, the (oneway) proxy abstraction known from RPC implementations has been used in several contexts to model a form of publish/subscribe interaction, where the addressing scheme is implicitly defined by the types of the subscribers. This is opposed to TPS, where the addressing scheme is implicitly given by the types of the events.

The division of the subscription space, if possible, according to several axes representing (static) filtering criteria such as subscriber, event, or even publisher types could lead to very interesting models. Also, would it be interesting to see what properties one could obtain by combining TPS, or any new hybrid model, with event correlation, i.e., adding a time axis to the space of possible subscriptions.

Languages and Abstractions for Distributed Programming

A long-term goal behind seeking for languages which enable library implementations of TPS without requiring additions to their very core, is to discover a set (or several sets) of language mechanisms that might not only enable a "clean" implementation of TPS, but maybe also the implementation of any distributed programming abstraction along the same lines, i.e., type safety, encapsulation, etc.

At the same time, we expect the collected experience to help us devising *future programming languages* embracing paradigms for distributed programming. Clues for this task could also be obtained by steering our previously mentioned research around the "mother of all publish/subscribe abstractions" towards identifying the "mother of all distributed programming abstractions", an extremely versatile abstraction which could be instantiated for distributed interaction schemes as diverse as RPC or shared spaces.

Interoperability

At a more short term, we are also looking at *existing programming languages*, in order to devise an interoperable event type system based on an EDL. The currently investigated solution attempts to promote structural conformance of obvents, motivated by the obviously further increased decoupling of participants.

Rather than extending every candidate language to obtain something similar to Java$_{PS}$ in the case of Java, we are more attracted by a solution which would leave the individual supported languages unchanged. The current approach consists in generating adapters according to a well-defined pattern (and mappings with small variances for the different languages), yet making these visible to the application developer. These adapters furthermore represent an ideal place to deal with structural conformance.

This research is strongly driven by the desire of convincing the distributed systems community that, in the context of publish/subscribe, interoperability and late binding seem to be no longer valid arguments against static typing.

Advanced Reliability

Last but not least, another field for long-term future research activity concerns reliability issues, such as persistence, but also transactions not considered in this dissertation. As shown by [TR00], there are several ways of integrating transactions with asynchronous event delivery, depending on the precise interaction model. Database systems supporting some form of publish/subscribe interaction nearly all provide such transactions, especially also in combination with message queueing.

This activity might also give more information on the links between general interaction paradigms, and fundamental problems in distributed computing. Just like a shared space can be used to provide an abstraction for mutual exclusion by exploiting the destructive `read()` primitive, one could imagine exploiting the straightforwardly appearing relationship between the group paradigm and the publish/subscribe paradigm. For instance, by the equivalence of the ever popular *consensus* problem [FLP85] and total order multicast (*atomic multicast*) [CT96], a publish/subscribe abstraction implementing totally ordered events can be used as a general programming abstraction for consensus.

Bibliography

[ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure Detection and Consensus in the Crash Recovery Model. *Distributed Computing*, 13(2):99–125, April 2000.

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

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Mahlki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG)*, pages 292–312, November 1992.

[AEM99] M. Altherr, M. Erzberger, and S. Maffeis. 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 '99)*, pages 43–53, October 1999.

[AFM97] O. Agesen, S.N. Freund, and J.C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 49–65, October 1997.

[Agh85] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* PhD thesis, University of Michigan, Computer and Communication Science, 1985.

[AOC$^+$88] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[ASS$^+$99] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–62, November 1999.

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review*, 26(2):92–109, April 1992.

[Bai75] N.T.J. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications (second edition).* Hafner Press, 1975.

[BC90] G. Bracha and W.R. Cook. 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 '90)*, pages 303–311, October 1990.

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

[BCC$^+$95] K.B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G.T. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BG93] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, pages 215–230, October 1993.

[BHJ$^+$87] A.P. Black, N. Hutchinson, E. Jul, H.M. Levy, and L. Carter. Distribution and Abstract Types in EMERALD. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BHL95] B. Blakeley, H. Harris, and J.R.T. Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development.* McGraw-Hill, 1995.

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

[BI93] A.P. Black and M.P. Immel. Encapsulating Plurality. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, pages 56–79, July 1993.

[Bir93] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

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

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

[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.

[BOW98] K.B. Bruce, M. Odersky, and Ph. Wadler. A Statically Safe Alternative to Virtual Types. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98)*, pages 523–549, July 1998.

[BPF97] K.B. Bruce, L. Petersen, and A. Fiech. Subtyping Is Not a Good "Match" for Object-Oriented Languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, pages 104–127, June 1997.

[BR97] G. Baumgartner and V.F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[Bri89] J.-P. Briot. 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 '89)*, pages 109–129, July 1989.

[BSvG95] K.B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*, pages 27–51, August 1995.

[BvR94] K.P. Birman and R. van Renesse. RPC Considered Inadequate. In *Reliable Distributed Computing with the Isis Toolkit*, pages 68–78. IEEE Computer Society Press, 1994.

[BW98] M. Büchi and W. Weck. Compound Types for Java. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 362–373, October 1998.

[Car86] L. Cardelli. The Amber Machine. In *Combinators and Functional Programming Languages*, volume 242 of *LNCS*, pages 48–70. Springer, 1986.

[Car93] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36:90–102, September 1993.

[Car95] L. Cardelli. A Language with Distributed Scope. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 286–297, August 1995.

[CCH$^+$89] P. Canning, W.R. Cook, W. Hill, W. Olthoff, and J.C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings 4th ACM International Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 273–280, September 1989.

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

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

[Cha98] D. Chatterton. *Dynamic Dispatch in Existing Strongly-Typed Languages.* PhD thesis, School of Computer Science & Software Engineering, Monash University, Australia, June 1998.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance Is Not Subtyping. In *Conference Record of the 17th ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 125–135, January 1990.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96)*, pages 285–299, October 1995.

[Chi00] S. Chiba. Loadtime Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 313–336, June 2000.

[CLCM00] C. Clifton, G.T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, pages 130–145, October 2000.

[CN91] B.J. Cox and A. Novabilsky. *Object Oriented Programming: an Evolutionary Approach.* Addison-Wesley, 1991.

[CNF98] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 10th IEEE International Conference on Software Engineering (ICSE '98)*, pages 261–270, April 1998.

[CNH99] M. Philippsen Ch. Nester and B. Haumacher. A More Efficient RMI for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 152–159, June 1999.

[Col99] M. Colan. InfoBus 1.2 Specification. Technical report, Sun Microsystems Inc., February 1999.

[Coo89] W.R. Cook. A Proposal for Making Eiffel Type-Safe. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP '89)*, pages 57–72, July 1989.

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

[CR97] P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 213–228. Springer, April 1997.

[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 2000)*, pages 219–227, July 2000.

[CS98] C. Cartwright and G. Steele. Compatible Genericity with Runtime Types for the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 201–215, October 1998.

[CT96] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DACE] DACE Distributed Asychronous Computing Environment. http://www.d-a-c-e.com.

[DEC94] DEC. *DECMessageQ: Introduction to Message Queuing*, April 1994.

[Dee91] S. Deering. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, 1991.

[DEG00] C.H. Damm, P.Th. Eugster, and R. Guerraoui. Abstractions for Distributed Interaction: Guests or Relatives? Technical Report DSC/2001/052, Swiss Federal Institute of Technology, Lausanne, June 2000.

[DGH$^+$87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 1–12, August 1987.

[DLS$^+$01] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 77–92, January 2001.

[EBGS01] P.Th. Eugster, R. Boichat, R. Guerraoui, and J. Sventek. Effective Multicast Programming in Large Scale Distributed Systems. *Concurrency and Computation: Practice and Experience*, 13(6):421–447, May 2001.

[EFGK01] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. Technical Report DSC/2001/004, Swiss Federal Institute of Technology, Lausanne, January 2001.

[EG00] P.Th. Eugster and R. Guerraoui. Type-Based Publish/Subscribe. Technical Report DSC/2000/029, Swiss Federal Institute of Technology, Lausanne, June 2000.

[EG01a] P.Th. Eugster and R. Guerraoui. Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 131–146, January 2001.

[EG01b] P.Th. Eugster and R. Guerraoui. Probabilistic Multicast. Technical report, Swiss Federal Institute of Technology, Lausanne, October 2001.

[EGD01] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

[EGH$^+$01] P.Th. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight Probabilistic Broadcast. In *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, pages 443–452, June 2001.

[EGS00] P.Th. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[EGS01] P. Th. Eugster, R. Guerraoui, and J. Sventek. Loosely Coupled Components. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, chapter 8. Kluwer, 2001.

[EKG01] P.Th. Eugster, P. Kouznetsov, and R. Guerraoui. $\Delta-$Reliable Broadcast. Technical report, Swiss Federal Institute of Technology, Lausanne, January 2001.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 317–326, October 1989.

[FG00] P. Felber and R. Guerraoui. Programming with Object Groups in CORBA. *IEEE Concurrency*, 8(1):48–58, January/March 2000.

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

[FJM$^+$96] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, pages 784–803, November 1996.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):217–246, 1985.

[For98] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface. Technical report, Message Passing Interface Forum, May 1998.

[GCLR92] R. Guerraoui, R. Capobianchi, A. Lanusse, and P. Roux. Nesting Actions through Asynchronous Message Passing: The ACS Protocol. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92)*, pages 170–184, June 1992.

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

[Gel89] D. Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE '89) Vol. II*, pages 20–27, June 1989.

[GG00] B. Garbinato and R. Guerraoui. An Open Framework for Reliable Distributed Computing. *ACM Computing Surveys*, 32(1), March 2000.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHvR$^+$97] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K.P. Birman. GSGC: An Efficient Gossip-Style Garbage Collection Scheme for Scalable Reliable Multicast. Technical Report TR97-1656, Cornell University, Computer Science, December 1997.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[Gol92] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California at Santa Cruz, December 1992.

[GR83] A.J. Goldberg and A.D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GS00] P. Gregono and M. Sakkinen. Copying and Comparing: Problems and Solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 226–250, June 2000.

[GS01] R. Guerraoui and A. Schiper. Genuine Atomic Multicast in Asynchronous Distributed Systems. *Theoretical Computer Science*, 254(1–2):297–316, March 2001.

[Gue99] R. Guerraoui. What Object-Oriented Distributed Programming Does not Have to Be, and what it May Be. *Informatik*, 2, April 1999.

[GvRB01] I. Gupta, R. van Renesse, and K.P. Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. In *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, pages 433–442, June 2001.

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

[HGM01] Y. Huang and H. Garcia-Molina. Publish/Subscribe in a Mobile Environment. In *2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE'01)*, 2001.

[HLS97] T. Harrison, D. Levine, and D.C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 184–200, October 1997.

[HMN$^+$00] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. In *Proceedings of the 5th IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 83–92, June 2000.

[HSC95] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of the 1995 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*, pages 328–341, August 1995.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[IBM95] IBM. *Smalltalk Tutorial.* http://www.smalltalksystems.com/references.htm, 1995.

[Ing86] D.H.H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 347–349, September 1986.

[ION96] IONA. *OrbixTalk Programming Guide.* IONA Technologies Ltd., Jul 1996.

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

[KGHK01] P. Kouznetsov, R. Guerraoui, S.B. Handurukande, and A.-M. Kermarrec. Reducing Noise in Gossip-Based Reliable Broadcast. In *Proceedings of the 20th IEEE Symposium On Reliable Distributed Systems (SRDS'01)*, October 2001.

[Kie97] Th. Kielmann. *Objective Linda: A Coordination Model for Object Oriented Parallel Programming.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, September 1997.

[KMG00] A.-M. Kermarrec, L. Massoulie, and A.J. Ganesh. Reliable Probabilistic Communication in Large-Scale Information Dissemination Systems. Technical Report MSR-TR-2000-105, Microsoft Research Cambridge, October 2000.

[KML93] D.G. Kafura, M. Mukherji, and G. Lavender. ACT++: A Class Library for Concurrent Programming in C++ Using Actors. *Journal of Object Oriented Programming*, pages 47–55, October 1993.

[KMMPN83] B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. Abstraction Mechanisms in the BETA Programming Language. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages (POPL '83)*, pages 285–298, January 1983.

[KMS98] G. Kirby, R. Morrison, and D. Stemple. Linguistic Reflection in Java. *Software - Practice and Experience*, 28(10):1045–1077, 1998.

[Koe99] P. Koenig. Messages vs Objects for Application Integration. *Distributed Computing*, 2(3):44–45, April 1999.

[KPHW89] G.E. Kaiser, S.S. Popovich, W. Hseush, and S.F. Wu. Melding Multiple Granularities of Parallelism. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP '89)*, pages 147–166, July 1989.

[KR95] B. Krishnamurthy and D.S. Rosenblum. Yeast: A General Purpose Event–Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[LAJ98] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Wide-Area Backbone Failures. In *Proceedings of the 29rd IEEE International Symposium on Fault-Tolerant Computing (FTCS '99)*, pages 278–285, June 1998.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LB98] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 36–44, October 1998.

[LBR96] K. Läufer, G. Baumgartner, and V.F. Russo. Safe Structural Conformance for Java. Technical Report CSD-TR-96-077, Department of Computer Sciences, Purdue University and West Lafayette, December 1996.

[LCD$^+$95] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A.C. Myers. Theta Reference Manual. Technical Report Programming Methodology Group Memo 88, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1995.

[Lea97a] D. Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[Lea97b] D. Lea. Design for Open Systems in Java. In *2nd International Conference on Coordination Models and Languages*, http://gee.cs.oswego.edu/dl/coord/, 1997.

[Lib01] J. Liberty. *Programming C#.* O'Reilly and Associates, Inc., July 2001.

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

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

[LLW99] T.J. Lehman, S.W. Mac Laughry, and P. Wyckoff. TSpaces: The Next Wave. In *Proceedings of the 32nd IEEE Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[LM99] M.-J. Lin and K. Marzullo. Directional Gossip: Gossip in a Wide Area Network. In *Proceedings of the 3rd European Dependable Computing Conference (EDCC-3)*, pages 364–379, September 1999.

[LMM00] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus Deterministically Constrained Flooding on Small Networks. In *Proceedings of the 14th International Conference on Distributed Computing Distributed Computing (DISC 2000)*, pages 253–267, October 2000.

[M01̈] G. Mühl. Generic Constraints for Content-Based Publish/Subscribe Systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS)*, September 2001.

[Mad99] O.L. Madsen. Semantic Analysis of Virtual Classes and Nested Classes. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, pages 114–131, November 1999.

[MAE+65] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, 1965.

[MBL97] A.C. Myers, J.A. Bank, and B. Liskov. Parameterized Types for Java. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 132–145, January 1997.

[Mey86] B. Meyer. Genericity versus Inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 391–405, September 1986.

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

[Mic97] Microsoft. *Microsoft Message Queuing Services*, 1997.

[Mil77] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computing Systems Sciences*, 17:348–375, December 1977.

[MK88] S. Matsuoka and S. Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 276–284, November 1988.

[MMMP90] O.L. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. 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 '90)*, pages 140–150, October 1990.

[MMSA91] L.E. Moser, P.M. Melliar-Smith, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS '91)*, pages 480–489, May 1991.

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

[MSS97] M. Mansouri-Samani and M. Sloman. GEM: A Generalized Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering*, 4(2):96–108, June 1997.

[Muc96] P. Muckelbauer. *Structural Subtyping in a Distributed Object System.* PhD thesis, Department of Computer Sciences, Purdue University, 1996.

[Nie87] O. Nierstrasz. Active Objects in Hybrid. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 243–253, December 1987.

[NN88] F. Nielson and H.R. Nielson. Two-Level Semantics and Code Generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[OAA$^+$00] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R.E. Strom, and D.C. Sturman. 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 2000)*, pages 185–207, April 2000.

[Obe00] R.J. Oberg. *Understanding & Programming COM+.* Prentice Hall, 2000.

[Obj99] ObjectSpace. *JGL - Generic Collection Library.* http://www.objectspace.com/jgl/, 1999.

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

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

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

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 58–68, December 1993.

[Ora99] Oracle. *Oracle8i Application Developer's Guide – Advanced Queuing*, 1999.

[ORO00] J. Orlando, L. Rodrigues, and R. Oliveira. Semantically Reliable Multicast Protocols. In *Proceedings of the 19th IEEE Symposium On Reliable Distributed Systems (SRDS'00)*, October 2000.

[OvRBX99] O. Ozkasap, R. van Renesse, K.P. Birman, and Z. Xiao. Efficient Buffering in Reliable Multicast Protocols. In *Proceedings of the 1st International COST264 Workshop on Networked Group Communication (NGC '99)*, pages 188–203, November 1999.

[OW97] M. Odersky and Ph. Wadler. Pizza into Java: Translating Theory into Practice. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 146–159, January 1997.

[Pit87] B. Pittel. On Spreading of a Rumor. *SIAM Journal of Applied Mathematics*, 47:213–223, 1987.

[PO93] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proceedings of the 12th International Conference on Entity-Relationship Approach*, pages 37–49, December 1993.

[Pol93] A. Polze. Using the Object Space: A Distributed Parallel make. In *Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 234–239, September 1993.

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

[Pow96] D. Powell. Group Communications. *Communications of the ACM*, 39(4):50–97, April 1996.

[PS97] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *Proceedings of the 27rd IEEE International Symposium on Fault-Tolerant Computing (FTCS '97)*, pages 309–313, June 1997.

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

[PTM96] J. Protić, M. Tomašević, and V. Milutinović. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology: Systems and Applications*, 4(2):63–79, August 1996.

[Rei91] M. Reiser. *The Oberon System.* ACM Press, 1991.

[Riv96] F. Rivard. Smalltalk: A Reflective Language. In *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection '96)*, pages 21–38, April 1996.

[RKF93] W. Rosenberry, D. Kenney, and G. Fisher. *OSF Distributed Computing Environment: Understanding DCE.* O'Reilly and Associates, Inc., 1993.

[Ros01] M. Roserens. Stock Trading with Distributed Asynchronous Collections. Master's thesis, Swiss Federal Institute of Technology, in collaboration with Lombard & Odier Co., March 2001.

[RSB$^+$98] D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In *Pattern Languages of Program Design 3*, chapter 17, pages 293–312. Addison-Wesley, 1998.

[RW96] A. Rowstron and A. Wood. Solving the Linda Multiple `rd` Problem. In *Proceedings of the 1st International Conference on Coordination Models and Languages*, pages 357–367. Springer, April 1996.

[RW97] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *6th European Software Engineering Conference/5th ACM Symposium on the Foundations of Software Engineering*, pages 344–360, September 1997.

[SA98] J.H. Solorzano and S. Alagic. Parametric Polymorphism for Java: A Reflective Solution. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 216–225, October 1998.

[SAB$^+$00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG2K)*, June 2000.

[Ses97] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* John Wiley & Sons, 1997.

[Sha86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS '86)*, pages 198–204, May 1986.

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

[SL95] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Silicon Graphics Inc., October 1995.

[SO95] D.D. Straube and M.T. Özsu. Query Optimization and Execution Plan Generation in Object-Oriented Data Management Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), April 1995.

[SOM94] C. Szyperski, S. Omohundro, and S. Murer. Engineering a Programming Language: The Type and Class System of Sather. In *Programming Languages and System Architectures*, volume 782 of *LNCS*, pages 208–227. Springer, March 1994.

[Sri95] R. Srinivasan. RFC 1831: Remote Procedure Call Protocol Specification Version 2. Technical report, Sun Microsystems, Inc., August 1995.

[SS00] Q. Sun and D.C. Sturman. A Gossip-Based Reliable Multicast for Large-Scale High-Throughput Applications. In *Proceedings of the 2000 IEEE International Conference on Dependable Systems and Networks (DSN 2000)*, pages 347–358, July 2000.

[Str97] B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Sun99] Sun. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*, 1999.

[Sun00a] Sun. *Java Core Reflection API and Specification*, 2000.

[Sun00b] Sun. *The Java Platform 1.3 API Specification*, 2000.

[Sun00c] Sun. *The Java Collections Framework*, 2000.

[SV97] D. Schmidt and S. Vinoski. Overcoming Drawbacks in the OMG Event Service. *SIGS C++ Report magazine*, 19(6), June 1997.

[Sys00] BEA Systems. *Reliable Queuing Using BEA Tuxedo: White Paper.* http://www.beasys.com/products/, 2000.

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

[TCKI00] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. A Class-Based Macro System for Java. In *Reflection and Software Engineering*, number 1826 in LNCS, pages 119–135. Springer, July 2000.

[Tho97] K.K. Thorup. Genericity in Java with Virtual Types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, pages 444–471, June 1997.

[Tho98] A. Thomas. Entreprise JavaBeans Technology: Server Component Model for the Java Platform. Technical report, Sun Microsystems Inc., December 1998.

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

[TR00] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, pages 308–330, 2000.

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

[TT99] K.K. Thorup and M. Torgersen. Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, pages 186–204, June 1999.

[TT01] H. Lam Th. Thai. *.NET Framework Essentials*. O'Reilly and Associates, Inc., June 2001.

[UM99] N. Uramoto and H. Maruyama. InfoBus Repeater: A Secure and Distributed Publish/Subscribe Middleware. In *International Workshops on Parallel Processing of the 28th IEEE International Conference on Parallel Processing (ICPP '99)*, pages 260–265, September 1999.

[US87] D. Ungar and R.B. Smith. The Power of Simplicity. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 227–241, October 1987.

[vR00] R. van Renesse. Scalable and Secure Resource Location. In *Proceedings of the 33rd IEEE Hawaii International Conference on System Sciences (HICSS-33)*, 2000.

[vRBM96] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[vRMH98] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Proceedings of the 2nd IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[vSHT99] M. v. Steen, Ph. Homburg, and A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, January-March 1999.

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM Object-Oriented Programming and Systems Messenger*, 1(1):7–87, August 1990.

[WEK97] D.A. Wallach, D.R. Engler, and M.F. Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. *IEEE/ACM Transactions on Networking*, 5(4):460–474, August 1997.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A High Performance Totally Ordered Multicast Protocol. In *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 33–54. Springer, 1995.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems Inc., November 1994.

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

[YT87] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 406–415, 1987.

[ZO01] M. Zenger and M. Odersky. Implementing Extensible Compilers. In *ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, June 2001.

List of Figures

List of Tables

Curriculum Vitae

Patrick Th. Eugster was born on October 7, 1973, in Mineola, Long Island, New York. At the age of 7, he moved near Zofingen, a small town in the german speaking part of Switzerland, where he obtained a high school degree in natural sciences. In 1998, he received his M.S. in Computer Science from the Swiss Federal Institute of Technology, Lausanne (EPFL). Since then he has been working as a research assistant and Ph.D. student, first in the Operating Systems Laboratory (LSE) of EPFL, and thereafter in the newly formed Distributed Programming Laboratory (LPD).

During his stay in the LSE, he has been involved in the European research project OpenDREAMS II (project 25262) where he represented EPFL in the role of technical manager, and has also acted several times as lecturer for postgraduate classes on reliable and object-based distributed programming. As member of the LPD, he has given lectures in classes on both distributed programming as well as object-oriented programming. Also, he co-initiated the DACE project, which served as test environment for his research focusing on reliable and object-based distributed systems and programming.