Book Title
Book Editors
IOS Press, 2003

Quality of Service in Publish/Subscribe Middleware[1]

Angelo CORSARO [b], Leonardo QUERZONI [a,2], Sirio SCIPIONI [a],
Sara TUCCI PIERGIOVANNI [a] and Antonino VIRGILLITO [a]
[a] *Universitá di Roma La "Sapienza"*
[b] *Selex SI - Roma*

Abstract. During the last decade the publish/subscribe communication paradigm gained a central role in the design and development of a large class of applications ranging from stock exchange systems to news tickers, from air traffic control to defense systems. This success is mainly due to the capacity of publish/subscribe to completely decouple communication participants, thus allowing the development of applications that are more tolerant to communications asynchrony. This chapter introduces the publish/subscribe communication paradigm, stressing those characteristics that have a stronger impact on the quality of service provided to participants. The chapter also introduce the reader to two widely recognized industrial standards for publish/subscribe systems: the Java Message Service (JMS) and the Data Distribution Service (DDS).

Keywords. Publish/Subscribe, Event-based Systems,

1. Introduction

Since the early nineties, anonymous and asynchronous dissemination of information has been a basic building block for many different distributed applications such as stock exchanges, news tickers, air-traffic control, industrial process control, etc.

Publish/Subscribe systems are nowadays considered a key technology for information diffusion. Each participant in a publish/subscribe communication system can play the role of a *publisher* or a *subscriber* of information. Publishers produce information in form of events, which are then consumed by subscribers. Subscribers can declare their interest on a subset of the whole information issuing subscriptions. Subscriptions are used to filter out part of the events produced by publishers.

The main semantical characterization of publish/subscribe is in the way events flow from publishers to subscribers: subscribers are not directly known by publishers, but rather they are indirectly addressed according to the content of events. This form of anonymity completely decouples publishers from subscribers, thus possibly allowing large scale deployments. Interactions between publishers and subscribers is mediated by

[1]This work was partially supported by a grant CINI-Finmeccanica on "QoS in information dissemination within network-centric architectures" and by the RESIST project, funded by the European Community.
[2]Correspondence to: Leonardo Querzoni, Via Salaria, 113 - 00198 Roma. Tel.: +39 06 4991 8480; Fax: +39 06 8530 0849; E-mail: querzoni@dis.uniroma1.it.

the publish/subscribe system, that, in general, is constituted by a set of nodes that coordinate among themselves in order to dispatch published events to all (and possibly only) interested subscribers.

Since publish/subscribe has been largely recognized as an effective approach for information diffusion, several publish/subscribe-based systems, both research contributions and commercial products have been presented and are actually used in many application contexts. From the research side, much work has been done in this field specifically by software engineering and distributed systems communities (focusing on scalability, efficient information delivery or efficient and expressive information matching). From the industrial side, relevant achievements are the widespread industrial standards that define semantics and interfaces for pub/sub middleware (Common Object Request Broker Architecture (CORBA) Event Service (CosEvent) [23], the CORBA Notification Service (CosNotification) [24], Java Message Service (JMS) [21] and, recently, Data Distribution Service (DDS) [19]). In both worlds, one important problem is related to the definition of quality of service (QoS) provision, defined as the guarantees that a pub/sub middleware can offer in terms of timeliness, reliability, availability etc. Market-ready solutions clearly must be able to provide QoS guarantees, for example in order to be deployed in mission critical applications. The definition and enforcement of QoS properties can be on the other hand a great inspiration for novel research contributions in this field.

The first part of this chapter gives the reader an overview of publish/subscribe systems, first introducing a general framework and then analyzing in details the models commonly used for subscriptions. Throughout this overview we focus on the definition of the very meaning of end-to-end QoS guarantees in a publish/subscribe system. Indeed, the complete decoupling between senders and receivers makes the exact semantics of the system not easily definable and subject to non-determinism. We identify the sources of such non-determinism and how to cope with it.

In the second part of the chapter the reader will be introduced to two important industrial standards for publish/subscribe middleware: the Java Message Service (JMS) [21] and the Data Distribution Service (DDS) [19]. JMS is a widely recognized standard for enterprise level messaging, targeted at applications such as application integration and large-scale data diffusion. Recently the Object Management Group (OMG) tried to sum up the characteristics of various proprietary publish/subscribe middleware products, to deliver a new standard for real-time oriented publish/subscribe; the result of this effort was the DDS specification. The two standards are presented by considering their general characteristics, their programming model and their QoS-related features. At the end of the chapter the reader should have gained an introductory knowledge about the ground where publish/subscribe middleware developers are today spending their efforts.

2. Framework

In this Section we define a general framework for publish/subscribe (pub/sub) systems. First we introduce the basic elements constituting a pub/sub system, then we discuss the semantics of the system.

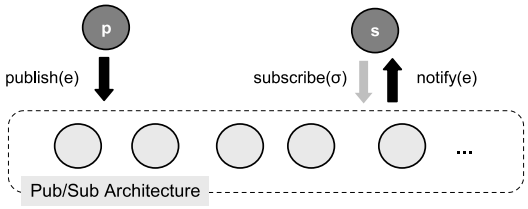

Figure 1. High-level interaction model of a publish/subscribe system with its clients (p and s indicate a generic publisher and a generic subscriber respectively).

2.1. Elements of a Publish/Subscribe System

A generic pub/sub communication system (often referred to in the literature as *Event Service* or *Notification Service*) is composed of a set of nodes distributed over a communication network. The clients of this system are divided according to their role into *publishers*, which act as producers of information, and *subscribers*, which act as consumers of information. Clients are not required to communicate directly among themselves but are rather *decoupled*: the interaction takes place through the nodes of the pub/sub system, that coordinate themselves in order to *route* information from publishers to subscribers. Participants' decoupling is a desirable characteristic in a communication system as applications can be easily developed just ingoring issues such as synchronization or direct addressing of subscribers.

Operationally, the interaction between client nodes and the pub/sub system takes place through a set of basic operations that can be executed by clients on the system and vice-versa (Figure 1). A publisher submits a piece of information e (i.e., an event) to the pub/sub system by executing the `publish(`e`)` operation. Commonly, an event is structured as a set of attribute-value pairs. Each attribute has a *name*, a simple character string, and a *type*. The type is generally one of the common primitive data types defined in programming languages or query languages (e.g. integer, real, string, etc.). On the subscribers' side, interest in specific events is expressed through *subscriptions*. A subscription σ is a filter over a portion of the event content (or the whole of it), expressed through a set of constraints that depend on the subscription language. A subscriber installs and removes a subscription σ from the pub/sub system by executing the `subscribe(`σ`)` and `unsubscribe(`σ`)` operations respectively.

We say that an event e $matches$ a subscription σ if it satisfies all the declared constraints on the corresponding attributes. The task of verifying whenever an event e matches a subscription σ is called $matching$.

2.2. Subscription Models

Various ways for specifying the subscribers' interest led to distinct variants of the pub/sub paradigm. The subscription models that appeared in the literature are characterized by their expressive power: highly expressive models offer to subscribers the possibility to precisely match their interest, i.e. to receive only the events they are interested in. In this section we briefly review the most popular pub/sub subscription models.

Topic-based Model Events are grouped in topics, i.e. a subscriber declares its interest for a particular topic to receive all events pertaining to that topic. Each topic corresponds to a logical channel ideally connecting each possible publisher to all interested subscribers. For the sake of completeness, the difference between channels and topics is that topics are carried within an event as a special attribute. Thanks to this coarse grain correspondence, either network multicast facilities or diffusion trees, one for each topic, can be used to disseminate events to interested subscribers.

The topic-based model has been the solution adopted in all early pub/sub incarnations. Examples of systems that fall under this category are TIB/RV [25], SCRIBE [8], Bayeux [31] and the CORBA Notification Service [24].

The main drawback of the topic-based model is the very limited expressiveness it offers to subscribers. A subscriber interested in a subset of events related to a specific topic receives also all the other events that belong to the same topic. To address problems related to low expressiveness of topics, several solutions are exploited in pub/sub implementations. For example, the topic-based model is often extended to provide hierarchical organization of the topic space, instead of a simple flat structure (such as in [1,25]). A topic B can be then defined as a sub-topic of an existing topic A. Events matching B will be received by all clients subscribed to both A and B. Implementations also often include convenience operators, such as wildcard characters, for subscribing to more than one topic with a single subscription[1]. Another method for enhancing expressiveness of the topic-based model is the *filtered-topic* variant [24,21], where a further filtering phase is performed once the message is received based on the content of the message. Messages that does not satisfy the filter are not delivered to the application.

Content-based Model Subscribers express their interest by specifying conditions over the content of events they want to receive. In other words, a subscription is a query formed by a set of constraints composed through disjunction or conjunction operators. Possible constraints depend on the attribute type and on the subscription language. Most subscription languages comprise equality and comparison operators as well as regular expressions [7,28,16]. The complexity of the subscription language obviously influences the complexity of matching operation. For this reason it is not common to have subscription languages allowing queries more complex than those in conjunctive form (examples are [5,4]). A complete specification of content-based subscription models can be found in [22]. Examples of systems that fall under the content-based category are Gryphon [20], SIENA [29], JEDI [12], LeSubscribe [27], Hermes [26], Elvin [28].

In content-based publish/subscribe, events are not classified according to some predefined criterion (i.e., topic name), but rather according to properties of the events themselves. As a consequence, the correspondence between publishers and subscribers is on a per-event basis. The difference with a filtered-topic model is that events that not match a subscriber can be filtered out in any point in the system, not only on the receiver, thus possibly saving network resources. For these reasons, the higher expressive power of content-based pub/sub comes at the price of a higher resource consumption needed to calculate for each event the set of interested subscribers [6,14].

[1] For the sake of completeness, we point out that the word *subject* can be used to refer to hierarchical topics instead of being simply a synonymous for topic. Analogously, *channel-based* is sometimes [23] used to refer to a flat topic model where the topic name is not explicitly included in the event.

Type-based In the type-based [15] pub/sub variant events are actually objects belonging to a specific type, which can thus encapsulate attributes as well as methods. With respect to simple, unstructured models, Types represent a more robust data model for application developer, enforcing type-safety at the pub/sub system, rather than inside the application. In a type-based subscription the declaration of a desired type is the main discriminating attribute. That is, with respect to the aforementioned models, type-based pub/sub sits itself somehow in the middle, by giving a coarse-grained structure on events (like in topic-based) on which fine-grained constraints can be expressed over attributes (like in content-based) or over methods (as a consequence of the object-oriented approach).

Concept-based The underlying implicit assumptions within all the above-mentioned subscription models is that participants have to be aware of the structure of produced events, both under a syntactic (i.e., the number, name and type of attributes) and a semantic (i.e., the meaning of each attribute) point of view. Concept-based addressing [11] allows to describe event schema at a higher level of abstraction by using ontologies, that provide a knowledge base for an unambiguous interpretation of the event structure, by using metadata and mapping functions.

XML Some research works [9,10,30] describe pub/sub systems supporting a semistructured data model, typically based on XML documents. XML is not merely a matter of representation but differs in the fact that introduces the possibility of hierarchies in the language, thus differentiating from a flat content-based model in terms of an added flexibility. Moreover, it provides natural advantages such as interoperability, independence from implementation and extensibility. As a main drawback, matching algorithms for XML-based language require heavier processing.

Location-awareness Pub/Sub systems used in mobile environments typically require the support for location-aware subscriptions. For example, a mobile subscriber can query the system for receiving notifications when it is in the proximity of a specific location or service. Works describing various forms of location-aware subscriptions are [18,30]. The implementation of location-aware subscriptions requires the pub/sub system the ability to monitor the mobility of clients.

2.3. Semantics of a Publish/subscribe System

In the following we intend to characterize the general semantics of a pub/sub system in terms of three properties stating the exact behavior of any pub/sub implementation[2]. This is critical for understanding the subtleties hidden behind the definition of the expected QoS offered by a pub/sub system and for highlighting what are the aspects of the system that are influential for it. We first consider two parameters that respectively take into account (i) non-instantaneous effects of subscribe/unsubscribe operations and (ii) the non-instantaneous diffusion of an event to the interested subscribers after a publish operation executed by a publisher. These parameters model the time required for the internal processing in the system and the network delay elapsed to route subscriptions and notifications, in a distributed implementation.

Indeed, when a process issues a subscribe/unsubscribe operation, the pub/sub system is not immediately aware of the occurred event. In other words, at an abstract level, the

[2]The discussion is here presented informally. A formalization of the pub/sub semantics can be found in [3]

registration (resp. cancellation) of a subscription takes a certain amount of time, denoted as T_{sub}, to be stored into the system. This time encompasses for example the update of the internal data structures of the pub/sub system and the network delay due to the routing of the subscription among all the entities constituting the system. Analogously, as soon as a publication is issued, the pub/sub architecture performs a *diffusion* of the information in order to reach the set of interested subscribers. This operation takes a certain amount of time during which the system computes and issues notify operations to interested subscribers, i.e. diffusion of events takes a non-zero time and is represented by a parameter T_{pub}.

The characterization of the exact behavior of the system is actually not obvious as (i) the interest of a subscriber is a dynamic dimension and (ii) the notification of an event can be issued to a subscriber at any time during the diffusion interval of the event itself. Then, semantics of a pub/sub system can be expressed by the following three properties:

- *Safety (Legality)*: a subscriber cannot be notified for an information it is not interested in.
- *Safety (Validity)*: a subscriber cannot be notified for an event that has not been previously published.
- *Liveness*: The delivery of a notification for an event is guaranteed only for those subscribers that subscribed at a time at least T_{sub} before the event was published and maintain their subscriptions stable for the entire time T_{pub} taken by the event's dissemination.

Safety properties describe *facts* that cannot happen during system execution, while Liveness gives a precise definition of which subscribers must be surely notified about an event. Obviously the longer a subscription remains stable in the system (i.e., it is *durable*), the higher its probability of meeting all the events, despite T_{pub}. The Liveness property can be extended by considering the possibility for the pub/sub system to persistently store events for a finite, non-zero amount of time, denoted as Δ. Persistence is exploited in distributed pub/sub implementations to provide reliable delivery of events through retransmission, or to allow notification of an event also to subscribers that subscribe *after* the event has been published. A revised definition of Liveness that take into account event persistence is:

- *Liveness (with persistent events)*: The delivery of a notification is guaranteed only for those subscribers that subscribed at a time at most $\Delta - T_{sub}$ after the event is published and maintain their subscriptions stable in the interval $[T_s + T_{sub}, max(T_s + T_{sub} + T_{pub}, T_e + T_{pub})]$.

where T_s and T_e are the times at which the subscription and the event were issued respectively.

3. Quality of Service in Publish/Subscribe Systems

Given the above definitions we can easily see that when considering end-to-end QoS characteristics in a pub/sub system one cannot set aside the effect of decoupling between senders and receivers, which is the main peculiar feature of the pub/sub paradigm. The

lack of a direct producer/consumer relationship makes the definition and enforcement of any end-to-end QoS policy very hard. Decoupling can introduce in several senses a *non-deterministic* behavior, meaning that the exact behavior of the system is difficult to specify, enforce and control. We give examples of how non-determinism can act over three fundamental aspects of QoS and security, namely reliable message delivery, timely delivery and trust relationship.

3.1. Reliable delivery

Reliable delivery of an event means determining the subscribers that have to receive a published event, as stated by the liveness property introduced in the previous section, and delivering the event to all of them. Event processing in the publish/subscribe infrastructure results in the event itself traveling several network hops, where each routing hop is potentially a source of non-determinism due to transmissions over asynchronous WAN channels or temporary node overloading. This can lead the value of T_{pub} to grow indefinitely, leading, from our definition of liveness, to a reduced probability of delivery of the notification to all the intended subscribers (*notification loss* [2]).

Persistence of events, durability of subscriptions and event retransmission can help to reduce the non-deterministic behavior, providing higher reliability in delivery. In general, the more an even remains in the system, the less non-determinism is experienced, at the price of a higher memory occupation. For example, the effect of runs between publications and subscriptions is limited and also the sensitivity to small delays in both subscription and publication dissemination. Reduction of non-determinism increases the probability that an intended receiver will get the information. If the information is stored in a permanently persistent way (i.e. with infinite memory) or it is infinitely retransmitted, non-determinism is completely absent and this probability raises to one.

3.2. Timeliness

Real-time applications often require strict control over the time elapsed by a piece of information to reach all its consumers. They are typically deployed over dedicated infrastructures or simply managed environments where synchronous message delivery can be safely assumed. Even in a completely managed environment, a pub/sub infrastructure which decouples publishers and subscribers, can introduce non-determinism through routing anomalies and unpredictable processing delays at each node. In overall, where timeliness constraints must be enforced, the design of the pub/sub system should privilege point-to-point communications where decoupling is limited or totally absent. The drawback of this choice lies in the main benefit introduced by the decoupling, that is the higher scalability obtainable by delegating the infrastructure, rather than the publishers, to know all the subscribers and determine the recipients for each event. Designing a QoS-driven pub/sub system which at the same time can scale to massive sizes is one major challenge in this area, particularly important for future implementations of the DDS specification (see Section 5).

3.3. Security and trust

Security issues represent one major problem in pub/sub systems, only marginally addressed at present by both researchers and industry. Aside from the obvious problem of

granting access to the system only to authorized participants, an important aspect regards enforcing trust between publishers and subscribers. A subscriber wants to trust authenticity of the events it receives from the system, i.e. they has been generated by a trusty publisher and the information they contains have not been corrupted. On the system side, subscribers have to be trusted for what concerns the subscriptions they issue.

Since an event is in general delivered to several subscribers, the producer/consumer trust relationship that commonly occur in a point-to-point communication, in pub/sub system must involve multiple participants. Moreover, the fact that message traverses several infrastructure nodes during routing forces both publishers and subscribers to rely such intermediary nodes not to corrupt events, subscriptions or some of the participants' identities.

Designing trust measures implies knowing with certainty the identity of other participants and this is in clear contrasts with the anonymity which is at the base of pub/sub itself. Under the assumption of trust the decoupling can be preserved by using a solution like the one presented in [13], where trust between a publisher and each subscriber is enforced through a chain of trust relationships involving all the nodes in the infrastructure that are met on the event path. In other words, when forwarding a message (either an event or a subscription), an infrastructure node is also responsible for letting the trust relationship flow with the message.

In the most general case where one cannot assume the whole infrastructure to be trustworthy, the possibility of an event traveling potentially malicious networks or nodes should be taken into account. In [17] a solution to this scenario is proposed. The idea is to organize groups of trust in *scopes*, i.e. logical domains within the pub/sub infrastructure. The organization in scopes limits the visibility of publishers, subscribers, events and subscriptions within a single scope in order to allow each scope to be independent under the points of view of management, routing algorithm and so on. Since a scope isolates its participants from outside traffic it allows to relax the assumption of a fully trusted infrastructure to each single scope. [17] describes a method to add a new trusted node to an existing trusted scope, so that the assumption of completely trusted scope is preserved. If the node to be added can be reached only through one or more untrusted nodes the request is tunneled so that only encrypted information transits through the non-trusted part of the network.

4. Java Message Service

Java Message Service [21] is a standard promoted by Sun Microsystems to define a Java API, including a common set of interfaces and semantics, for the implementation of message-oriented middleware. It is part of the Java Enterprise Edition (J2EE) architecture since version 1.3. The compliance to the specification allows implementations from various vendors to be perfectly interoperable. In this way JMS guarantees a portable way for Java applications to exchange messages through products of different vendors.

Besides a message-centric publish-subscribe communication model, the JMS API also supports a *point-to-point* mode. With point-to-point, each application produces messages that are explicitly targeted toward a single receiver. A JMS implementation then represents a general-purpose *message oriented middleware* (MOM) that acts as an intermediary between heterogeneous applications, allowing to choose the communication mode that better suits the specific application needs.

JMS is specifically targeted at distributed enterprise systems, frequently presenting problems such as integration among heterogeneous components, management of complex workflows, dissemination of large-size data on a large scale and reliable data delivery. Those issues can be easily faced by means of a loosely coupled, flexible and standard communication mechanism such as a JMS MOM, that can effectively help in reducing development costs and time.

4.1. JMS Conceptual Model

The JMS conceptual model marks a clear separation between the point-to-point and the publish-subscribe models; nevertheless, in both cases, only non strongly typed messages are considered. Each message is characterized by a header (which includes message type, priority, etc.), by a set of extension of header metadata used to support, for example, compatibility with specific implementations and provider-specific properties, and a body which includes the application specific data core of the message. In the following we provide a characterization of entities that constitute the JMS conceptual model.

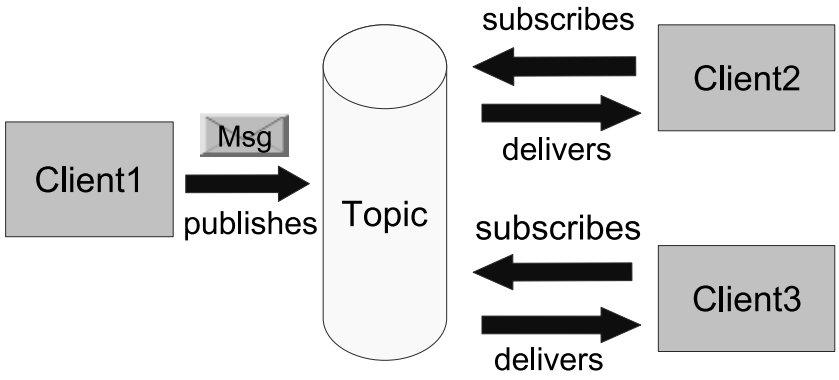

Figure 2. JMS Topic Model.

Topics. the JMS publish-subscribe API is based on *topics*. Publishers and Subscribers are anonymous and can dynamically publish and subscribe to various topics (see Figure 2). Applications can define reliability and QoS requirements for each topic.

Publishers and Subscribers. Publishers and Subscribers are the classes used for implementing producers and consumers for a topic. Multiple receivers can subscribe to the same topic and receive the same message. Topics, contrarily to queues, retain messages only as long as it takes to distribute them to current subscribers. The interaction is one-to-many and it has a timing dependency between senders and receivers: consumers receive only messages sent after their subscription and they must continue to be active in order to consume new messages (see Section 2.3). That is, events are not persistent. Non-determinism can be reduced by means of a *durable subscription*. Durable subscriptions provide the reliability of queues but nevertheless maintain the one-to-many interaction model. This aspect will be further analyzed in following section.

Subscriptions. In the JMS API subscriptions are topic-based. Applications requiring higher expressiveness can exploit a form of filtered-topic model, as defined in the *Message Selector* API, where filters can be applied directly on receiver-side to received messages. A message selector is an expression whose syntax is based on SQL92. It is evaluated when an attempt is made to receive a message, and messages that do not match the selection criteria are discarded. Message selectors only work on header fields and properties: body and content of the message cannot be used for selection. Contrarily to a pure content-based model, message filtering in JMS is executed only on receiver-side.

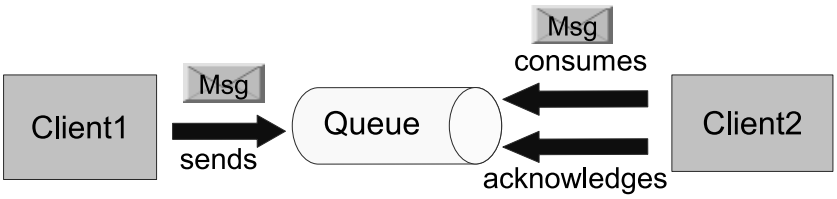

Figure 3. JMS Queue Model.

Point-to-point (Queues). The point-to-point model of JMS exploit *queues*, where messages are stored until they are consumed or expire. Senders and receivers have to bind to a queue to use it and once they subscribe they can start sending and retrieving messages (see Figure 3).

Messages are explicitly addressed to a queue, and analogously receivers extract messages directly from a queue. There is no timing dependency between the execution of send and receive operations: the receiver can retrieve a message even if it was not running when the sender sent it. Finally the consumer of a message can send an acknowledgment as a result of the delivery of the message to queue.

Discovery Another feature of JMS API is the ability to dynamically discover information related to topics: clients can explore topics and queues through a search on a centrally managed JNDI namespace.

4.2. JMS Programming model

A JMS application is composed from the following elements (Figure 4):

Administrated Objects. These are pre-configured objects that are created by administrators. They are of two types: `ConnectionFactory` and `Destination`. JMS clients access these objects through interfaces that have been standardized in the JMS specification, while the actual underlying technology strictly depends on the implementation. `ConnectionFactory` objects are used by clients to connect with a provider[3]. Each of these objects encapsulates a set of connection configuration parameters defined by an administrator. `Destination` objects are used by a sender to specify the target of a message it produces and by a re-

[3]JMS provider is a proprietary part of JMS application, which realizes the messaging system and provides administrative and control features.

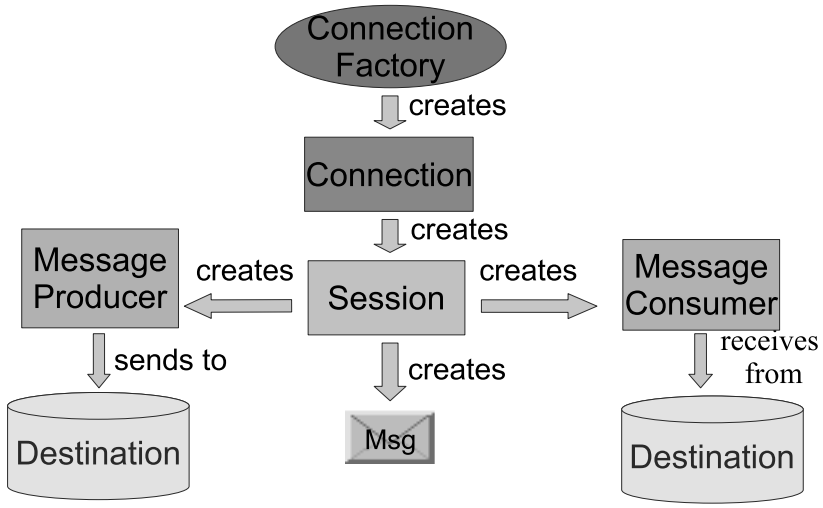

Figure 4. JMS Topic Model.

ceiver to specify the source of messages it consumes. In the point-to-point domain, `Destination` objects represent queues, while in the publish/subscribe domain they are called topics. Administrative and proprietary tools allow to create and to bind these two objects into a JNDI namespace. A JMS client can use JNDI to look up `ConnectionFactory` and `Destination` objects and establish a logical connection through the JMS provider.

Connections. Represent virtual connections to JMS providers. A connection is used to create sessions.

Sessions. Each `Session` object represents a single-threaded context for message producers, message consumers and messages. A session provides a transactional context where a set of sends and receives can be grouped in an atomic unit of work.

Message Producers and Consumers. Objects used for sending/receiving messages to/from destinations. Message production is asynchronous but JMS interface supplies two modality for message delivery: *synchronous* and *asynchronous*. Synchronous messages are delivered by calling the `receive` method. This method blocks the application until a message arrives or a timeout occurs. Asynchronous messages are consumed by creating a `message listener`. Its `onMessage` method is executed by the JMS provider when a message arrives at its destination.

4.3. Quality of Service

The only Quality of Service policy defined in the JMS specification is related to reliability. An application can require every message to be received once and only once or it rather can choose a more permissive (and generally more efficient) policy, allowing dropped and duplicated messages. JMS API specification provides various degree of reliability through various basic and advanced mechanisms.

4.3.1. Basic Reliability Mechanisms

The most interesting basic mechanisms are:

Specifying message persistence : a JMS application can specify that messages are persistent, thus ensuring that a message will not be lost in the event of a provider failure. Two delivery modes are defined in the JMS specification: *persistent* require JMS providers to log messages in a stable storage, while *non persistent* delivery mode does not require it.

Setting message priority levels : applications can set a message priority level; in this case the JMS provider will deliver urgent messages first. The JMS API provides methods to set priority levels for all messages sent by a producer, through the `setPriority` method of the `MessageProducer` interface, or to set priority level for specific messages, through *send* or *publish* methods of same interface.

Allowing messages to expire : in order to prevent duplicated messages an application can set an expiration time for a message. As in the previous case JMS API provides methods that allow to set a time to live counter for all messages produced from a publisher, or just a single one.

4.3.2. Advanced Reliability Mechanisms

The most advanced mechanism to provide reliable message delivery in the JMS specification is the creation of *durable subscriptions*. A durable topic subscription allows a subscriber to receive messages sent while it is not active. A durable subscription implements the reliability of queues in the publish/subscribe model. A durable subscription can have only one active subscriber at a time. When a durable subscriber registers a durable subscription, it specifies a unique identity by setting an ID for the connection and a topic and subscription name for the subscriber. Other subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber. The subscriber can be closed and reloaded, but the subscription continues to exist until the subscriber invokes the unsubscribe method. When the subscriber is reactivated the JMS provider sends it the stored messages.

Other features common in MOM products, like load balancing, resource usage control, and timeliness of messages, are not explicitly addressed in the JMS specification. Although recognized in the specification as fundamental for the development of robust messaging applications, they are considered provider-specific.

5. Data Distribution Service

The pub/sub paradigm is a natural match, and often a fundamental architectural building block, for a large class of real-time, mission, and safety critical application domains, such as industrial process control, air traffic control, defense systems, etc. These application domains are characterized by real-time information which flows from sensors to controllers and from controllers to actuators. The timeliness of data distribution is essential for maintaining the correctness and the safety of these systems, *i*.e., failing in timely delivering data could lead to instability which might result in threats to either infrastructures of human lives.

Historically, most of the pub/sub middleware standards such as the CosEvent [23], the CosNotification [24], and JMS [21], etc., as well as most proprietary solutions, have lacked the support needed by real-time, mission, and safety critical systems. The main limitations are typically due to the limited or non-existent support for Quality of Ser-

vice (QoS), and the lack of architectural properties which promote dependability and survivability, *e.g.*, lack of single point of failure.

Recently, in order to fill this gap, the OMG has standardized the DDS [19]. This standard gathers the experience of proprietary real-time pub/sub middleware solutions which had been independently engineered and evolved in niches, within the industrial process control, and in the defense systems applications domain. The resulting standard, which will be described in detail in the reminder of this Section, is based on a completely decentralized architecture, and provides an extremely rich set of configurable QoS.

Before proceeding with a detailed explanation of the DDS, it is worth mentioning that the standard defines two level of interfaces. At a lower level, it defines a Data Centric Publish Subscribe (DCPS) whose goal is to provide an efficient, scalable, predictable, and resource aware data distribution mechanism. Then, on top of the DCPS, it defines the Data Local Reconstruction Layer (DLRL), an optional interface which automates the reconstruction of data, locally, from updates received, and allows the application to access data as if it was local.

5.1. DDS Conceptual Model

The DDS conceptual model is based on the abstraction of a strongly typed Global Data Space (GDS) (see Figure 5), where publisher and subscriber respectively *write* (produce) and *read* (consume) data. In the reminder of this Section we will provide a precise characterization of the entities that constitute this global data space.

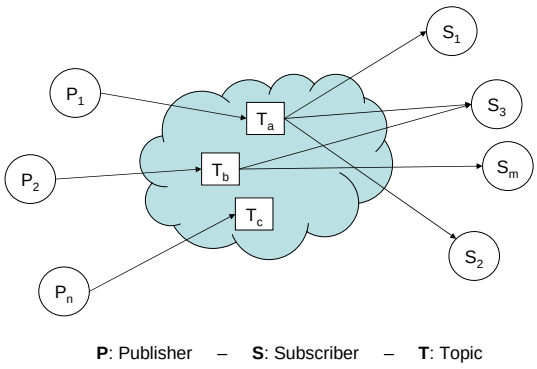

Figure 5. DDS Global Data Space.

Topic. A topic defines a type that can be legally written on the GDS. In the present standard, topics are restricted to be nonrecursive types defined by means of OMG Interface Definition Language (IDL). The DDS provides the ability to distinguish topics of the same type by relying on the use of a simple key. Finally, topics can be associated with specific QoS. From an applicative perspective, topics are the mean used by designer to define the application information model. The model supported by the DDS is not as powerful as that found in contemporary relational Data Base (DB)s, however it provides the ability to perform simple topic aggregation, as well as content based filtering.

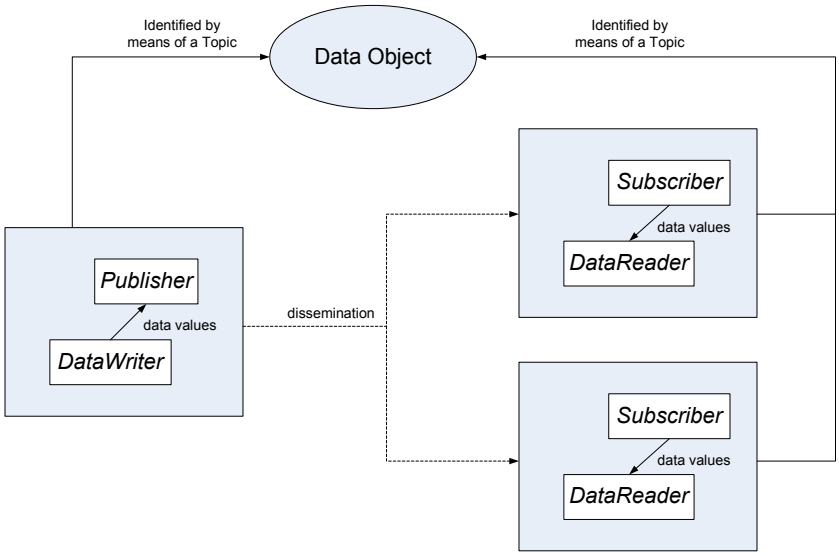

Figure 6. DDS ConceptualModel.

Publisher. Topics allow the definition of the application data model, as well as the association of QoS properties with it. On the other hand, publishers provide a mean of defining data sources. A publisher, can declare the intent of generating data with an associated QoS, and to *w*rite the data in the GDS. The publisher declared QoS has to be compatible with that defined by the topic. More specifically, as depicted in Figure 6, the DDS relies on a topic specific `DataWriter` which serves as a typed writer to the GDS. On the other hand, the `Publisher` encapsulate the responsibility associated with the dissemination of data in agreement with the required QoS.

Subscriber. Subscribers *r*ead topics in the global data space for which a matching subscription exist (the rules that define what represents a matching subscription are described below). The DDS relies on a topic specific `DataReader` which serves as a typed reader into the GDS. On the other hand, the `Subscriber` encapsulates the responsibility associated with the reception of data in agreement with the required QoS.

Subscription. A subscription is the logical operation which glues together a subscriber to its matching publishers. In the DDS a matching subscription has to satisfy two different kind of conditions. One set of conditions relate to concrete features of the topic, such as its type, its name, its key, its actual content. The other set of conditions relate to the QoS. More specifically, the DDS provides a subscription scheme which is more general than the typical topic-based model described in Section 2.2 as it also allows for content based subscription – a subset of Structured Query Language (SQL) is used for specifying subscription filters. Regarding the QoS, the matching follows an requested/offered model in which the requested QoS has to be the same, or weaker, then the offered. As an example, a matching subscription for a topic which is distributed reliably, can be requesting the topic to be distributed either reliably or as best effort.

Discovery. Another key feature at the foundation of DDS is that all information needed to establish a subscription is discovered automatically, and, in a completely distributed manner. The DDS discovery service, finds-out and communicates the properties of the GDS's participants, by relying on special topics and on the data dissemination capability provided by the DDS.

Finally, for sake of completeness, it is worth pointing out that the DDS supports the concept of domains. A domain allows to administratively separate and confine the distribution of different data flows. A DDS entity can belong to different domains, however data cannot flow across domains.

5.2. DDS Programming Model

Now that we have seen what are the core concepts at the foundation of DDS, we are ready to move to its programming model. Figure 7, contains an Unified Modeling Language (UML) diagram which represents the core DDS Application Programming Interface (API) in terms of its key classes and their relationships.

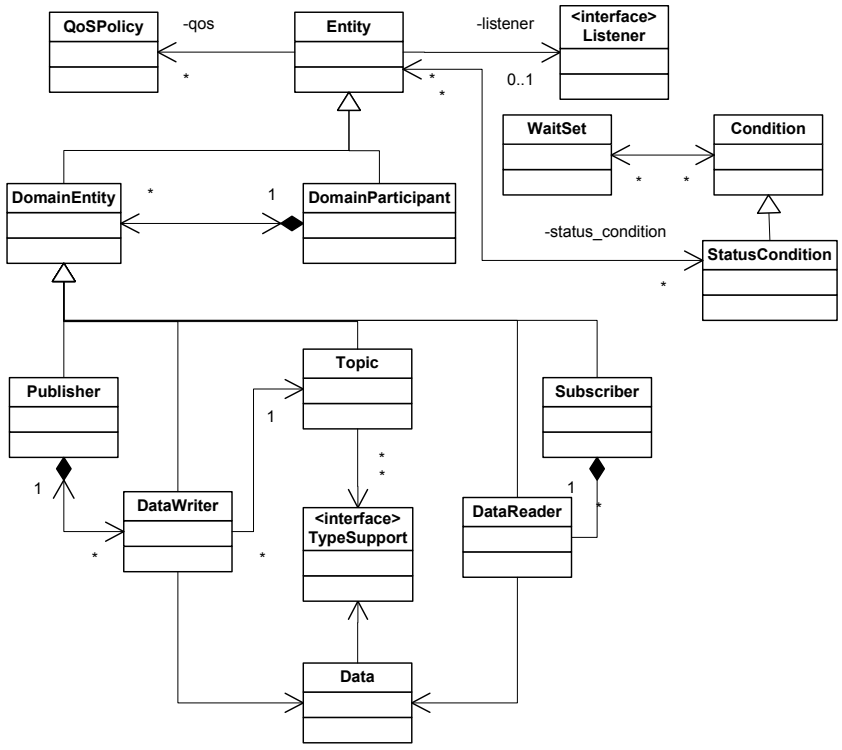

Figure 7. DDS Programming Model.

From Figure 7 it is worth noticing how the DDS API is mostly based on a rooted hierarchy at the base of which we find the `Entity` class. This class, by means of the association with the `QoSPolicy` class, defines the basic mechanisms for associating QoS with DDS entities. At the same time, with the association with the `Listener` and the `StatusCondition` classes define the two interaction model supported by the DDS API – the reactive and selective interaction model. The reactive model is supported by the `Listener` class. Instances of this class can be registered with any kind of DDS entity to receive callbacks on specific events, such as data being available for being read, etc. On the other hand, the selective model is supported by the `StatusCondition` class. Instances of this class can be used in a way similar to the UNIX `select` system call to poll or wait on specific conditions.

The `DomainParticipant` represents the local membership to a specific domain. Only publisher and subscribers belonging to the same domain can communicate. The `DomainEntity` exists essentially to enforce the fact that `DomainParticipant` cannot be nested. Finally, the diagram shows the classes defined by the DDS standard in order to write and read data from the GDS, *i*.e., `Publisher`, `Subscriber`, `DataWriter`, etc.

5.3. Quality of Service

One of the key distinguishing features of the DDS when compared to other pub/sub middleware is its extremely rich QoS support. By relying on a rich set of QoS policies, the DDS gives the ability to control and limit (1) the use of resources, such as, network bandwidth, and memory, and (2) many non functional properties of the topics, such as, persistence, reliability, timeliness, etc. In the reminder of this Section we will provide an overview of the most interesting QoS defined by the DDS classifying them with respect to the aspect they allow to control.

Resources

The DDS defines a specific QoS policy to control the resources which can be used to meet requested QoS on data dissemination. Below are reported the most relevant QoS policies which allow to control computing and network resources.

- The `RESOURCE_LIMITS` policy allows to control the amount of message buffering performed by a DDS implementation.
- The `TIME_BASED_FILTER` allows applications to specify the minimum inter-arrival time between data samples. Samples which are produced at a faster pace are not delivered. This policy allows to control both network bandwidth as well as memory and processing power for those subscribers which are connected over limited bandwidth networks and which might also have limited computing capabilities.

The DDS provides other means to control the resources consumed, however, these will be presented below as they also have an impact on application visible properties of data.

Data Timeliness

The DDS provides a set of QoS policies which allow to control the timeliness properties of distributed data. Specifically, the supported QoS are described below.

- The `DEADLINE` QoS policy allows application to define the maximum inter-arrival time for data. Missed deadline can be notified by `Listeners` (see Figure 7).
- The `LATENCY_BUDGET` QoS policy provides a means for the application to communicate to the middleware the level of urgency associated with a data communication. Specifically, the latency budget specifies the maximum amount of time that should elapse from the instant in which the data is written to the instant in which the data is placed in the queue of the associated readers.

Data Availability

The DDS provides the following QoS policies which allow to control the data availability.

- The `DURABILITY` QoS policy provides control over the lifetime of the data written on the GDS. At one extreme it allows the data be configured to be volatile, at the other it allows to have data persistency. It is worth noticing that transient and persistent data enables time decoupling between the writer and the reader by making the data available for late joining reader, in the case of transient data, or even after the writer has left the GDS, for persistent data.
- The LIFESPAN QoS policy allows to control the interval of time for which a data sample will be valid. The default value is infinite.
- The `HISTORY` QoS policy provides a mean to control the number of data samples, *i*.e., subsequent write of the same topic, have to be kept available for the readers. Possible values are the last, the last n samples, or all the samples.

Data Delivery

The DDS provides several QoS which allow to control how data is delivered and who is allowed to write a specific topic. More specifically the following QoS policies are defined.

- The `RELIABILITY` QoS policy allows application to control the level of reliability associated with data diffusion. The possible choices are reliable and best-effort distribution.
- The `DESTINATION_ORDER` QoS policy allows to control the order of changes made by publishers to some instance of a given topic. Specifically the DDS allows different changes to be ordered according to the source or the destination timestamp.
- The `OWNERSHIP` QoS policy allows to control the number of writers permitted for a given topic. If configured as exclusive, then it indicates that a topic instance can be owned and thus written by a single writer. The ownership of a topic is controlled by means of another QoS policy, the `OWNERSHIP_STRENGTH`. This additional policy makes it possible to associate a numerical strength to writers, so that the owner of a topic is defined to be the one available with the highest strength. If the `OWNERSHIP` QoS policy is configured as shared then multiple writer can concurrently update a topic. The concurrent changes will be ordered according to the `DESTINATION_ORDER` policy.

In addition to the QoS policies defined above, the DDS provides some mean of defining and distributing bootstrapping information by means of the USER_DATA, TOPIC_DATA and GROUP_DATA. These policies apply at different level, as it can be guessed by the name, and are distributed by means of built-in topics.

References

[1] S. Baehni, P. Th. Eugster, and R. Guerraoui. Data-aware multicast. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN 2004)*, pages 233–242, 2004.

[2] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. Measuring notification loss in publish/subscribe communication systems. In *Proceedings of the 10th International Symposium Pacific Rim Dependable Computing (PRDC '05)*, 2004.

[3] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. On the modelling of publish/subscribe communication systems. *Concurrency and Computation: Practice and Experience*, 17(12):1471–1495, 2005.

[4] S. Bittner and A. Hinze. On the benefits of non-canonical filtering in publish/subscribe systems. In *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'05)*, 2005.

[5] A. Campailla, S. Chaki, E. M. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of The International Conference on Software Engineering*, pages 443–452, 2001.

[6] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 219–227, 2000.

[7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowston. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[9] R. Chand and P. Felber. Xnet: A reliable content-based publish/subscribe system. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004)*, pages 264–273, 2004.

[10] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Parallel Processing, 11th International Euro-Par Conference (Euro-par 2005)*, pages 1194–1204, 2005.

[11] M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, August 2002.

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

[13] I. Dionysiou, D. Frincke, D. E. Bakken, and C. Hauser. Actor-oriented trust. Technical Report EECS-GS-006, School of Electrical Engineering and Computer SCience, Washington State University, Pullman, WA, USA, 2005.

[14] P.Th. Eugster, P. Felber, R. Guerraoui, and S.B. Handurukande. Event Systems: How to Have Your Cake and Eat It Too. In *Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[15] P.Th. Eugster, R. Guerraoui, and Ch.H. Damm. On Objects and Events. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2001.

[16] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001)*, pages 115–126, 2001.

[17] L. Fiege, A. Zeidler, A. Buchmann, R. Kilian-Kehr, and G. Muhl. Security aspects in publish/subscribe systems. In *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems*, 2004.

[18] Ludger Fiege, Felix C. Gärtner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, pages 103–122, 2003.

[19] Object Management Group. Data distribution service for real-time systems specification, 2002.

[20] Gryphon Web Site. http://www.research.ibm.com/gryphon/.

[21] Sun Microsystems Inc. Java message service api rev 1.1, 2002.

[22] G. Muhl. Generic Constraints for Content-Based Publish/Subscribe. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS)*, 2001.

[23] Object Management Group. CORBA event service specification, version 1.1. OMG Document formal/2000-03-01, 2001.

[24] Object Management Group. CORBA notification service specification, version 1.0.1. OMG Document formal/2002-08-04, 2002.

[25] B. Oki, M. Pfluegel, A. Siegel, and D. Skeen. The information bus - an architecture for extensive distributed systems. In *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, December 1993.

[26] P. Pietzuch and J. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2003.

[27] R. Preotiuc-Pietro, J. Pereira, F. Llirbat, F. Fabret, K. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Cairo, Egypt, 2000.

[28] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2K, Canberra, Australia*, June 2000.

[29] SIENA Web Site. http://www.cs.colorado.edu/users/carzanig/siena/.

[30] T. Sivaharan, G. Blair, and G. Coulson. GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. In *Proceedings of DOA 2005*, 2005.

[31] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.