OMG Data-Distribution Service: Architectural Overview

Gerardo Pardo-Castellote, Ph.D.

Real-Time Innovations, Inc.

gerardo@rti.com

Abstract

The OMG Data-Distribution Service (DDS) is an emerging specification for publish-subscribe data-distribution systems. The purpose of the specification is to provide a common application-level interface that clearly defines the data-distribution service. The specification describes the service using UML, thus providing a platform-independent model that can then be mapped into a variety of concrete platforms and programming languages.

The OMG DDS attempts to unify the common practice of several existing implementations [2, 5] enumerating and providing formal definitions for the QoS (Quality of Service) settings that can be used to configure the service.

This paper introduces the OMG DDS specification, describes the main aspects of the model, QoS settings, and gives examples of the communication scenarios it supports.

1 Introduction

The organization of the information exchange between modules is fundamental to publish-subscribe (PS) systems. The PS model connects anonymous information producers (publishers) with information consumers (subscribers). The overall distributed application (the PS system) is composed of processes, each running in a separate address space possibly on different computers. We will call each of these processes a "***participant***". A participant may simultaneously publish and subscribe to information. The defining aspect of a PS system is the decoupling in space, time, and flow between publishers and subscribers [1].

The information transferred by data-centric communications can be further classified into: **Signals**, **Streams**, and **States**. **Signals** represent data that is continuously changing (such as the readings of a sensor). Signals can often be sent best-efforts. **Streams** represent snapshots of the value of a data-object that must be interpreted in the context of previous snapshots. Streams often need to be sent reliably. **States** represent the state of a set of objects (or systems) codified as the most current value of a set of data attributes (or data structures). The state of an object does not necessarily change with any fixed period. Fast changes may be followed by long intervals without change. Consumers of "state data" are typically interested in the most current state. However, as the state may not change for a long time, the middleware may need to ensure that the most current state is delivered reliably. In other words, if a value is missed, then it is not always acceptable to wait until the value changes again.

The goal of the DDS specification is to facilitate the efficient distribution of data in a distributed system. Participants using DDS can 'read' and 'write' data efficiently and naturally[1] with a typed interface. Underneath, the DDS middleware will distribute the data so that each reading participant can access the 'most-current' values. In effect, the service creates a global "data space" that any participant can read and write. It also creates a name space to allow participants to find and share objects.

DDS targets real-time systems; the API and QoS are chosen to balance predictable behavior and implementation efficiency/performance[2]. We will note some of these tradeoffs in this paper.

The DDS specification describes two levels of interfaces:

- A lower Data-Centric Publish-Subscribe (DCPS) level that is targeted towards the efficient delivery of the proper information to the proper recipients
- An optional higher Data-Local Reconstruction Layer (DLRL) level, which allows for a simpler integration into the application layer.

This paper focuses on the DCPS layer. Figure 1 illustrates the overall model.

[1] Here "naturally" means that the interface should be similar to the one used to read/write local variables.

[2] For example, the DDS API requires each process to pre-declare the data it will generate as well as the data it wants to consume.

Communication is accomplished with the aid of the following entities: ***DomainParticipant, DataWriter, DataReader, Publisher, Subscriber,*** and ***Topic***. All these classes extend ***DCPSEntity***, representing their ability to be configured through QoS policies, be notified of events via listener objects, and support conditions that can be waited upon by the application. Each specialization of the ***DCPSEntity*** base class has a corresponding specialized listener and a set of ***QoSPolicy*** values that are suitable to it.

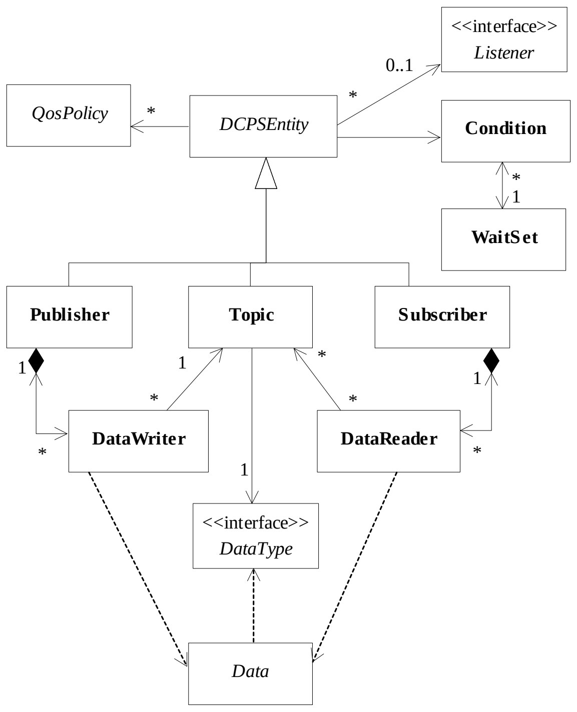

Figure 1 Overall DCPS model

Publisher represents the objects responsible for data issuance. A Publisher may publish data of different data types. A ***DataWriter*** is a typed facade to a publisher; participants use DataWriter(s) to communicate the value of and changes to data of a given type. Once new data values have been communicated to the publisher, it is the Publisher's responsibility to determine when it is appropriate to issue the corresponding message and to actually perform the issuance (the Publisher will do this according to its QoS, or the QoS attached to the corresponding DataWriter, and/or its internal state).

A ***Subscriber*** receives published data and makes it available to the participant. A Subscriber may receive and dispatch data of different specified types. To access the received data, the participant must use a typed ***DataReader*** attached to the subscriber.

The association of a ***DataWriter*** object (representing a publication) with ***DataReader*** objects (representing the subscriptions) is done by means of the ***Topic***. A ***Topic*** associates a name (unique in the system), a data type, and QoS related to the data itself. The type definition provides enough information for the service to manipulate the data (for example serialize it into a network-format for transmission). The definition can be done by means of a textual language (e.g. something like “float x; float y;”) or by means of an operational “plugin” that provides the necessary methods.

DCPS can also support content-based subscriptions by means of a filter (see Section 5). This is an optional feature because content-based filtering can be computationally intensive and introduce hard-to-predict delays. However, recent research [3] has proven that there are efficient algorithms to address this problem; and in any case, the application can restrict the filters to Topics where predictable distribution is not an issue.

DDS differs from “enterprise” publish-subscribe systems (such as Rendezvous [7], Vitrias’s architecture [6], SmartSockets [8] and JMS [11]) in its binding of ***Topic*** to a data-type. Topic is therefore more than a “routing” label. This coupling (with the additional QoS settings) enables implementation optimizations such as pre-allocating the resources needed to send or receive a ***Topic***.

The DCPS layer is composed of five modules: Infrastructure, Topic, Publication, Subscription, and Domain:

- The Infrastructure Module contains the ***DCPSEntity***, ***QosPolicy***, ***Listener*** , ***Condition***, and ***WaitSet*** classes. These abstract classes support two interaction styles: notification-based and wait-based. They implement interfaces that are refined by the other modules.
- The Topic-Definition Module contains the ***Topic*** and the ***TopicListener*** classes and more generally all that is needed by the application to define data-types, create topics, and attach QoS policies to them.
- The Publication Module contains the ***Publisher***, the ***DataWriter*** and the ***PublisherListener*** classes, and more generally all that is needed on the publication side.
- The Subscription Module contains the ***Subscriber***, the ***DataReader*** and the ***SubscriberListener*** classes, and more generally all that is needed on the subscription side.
- The Domain module contains the class ***DomainParticipantFactory*** (not shown in the figure) that acts as an entry-point for the service, as well as the class ***DomainParticipant*** that is a container for the other objects.

Details on each of these modules can be found in the OMG submission [10]. Here we will focus on a few aspects of the DDS that illustrate its suitability for data-centric communications: the identification of data-objects, the read and write access, the quality of service settings, and the interaction styles.

2 Identification of data-objects

A distinguishing aspect of publish-subscribe systems is the mechanism that producers use to identify the information published and subscribers use to specify the information they want. Application-defined strings (sometimes called topics or subjects) are a common minimalist approach. For event-distribution such strings combined with filters that that can operate on the contents or additional attributes added to the events may be sufficient. This is the approach taken, for example, by the CORBA Notification Service [9].

However, in data-centric systems, the information exchanges refer to values of an imaginary global data object. Given that new values typically override prior values, both application and middleware need to identify the actual instance of the "Global data object" the value applies to. In other words, a publisher writing the value of a data-object must have the means to indicate uniquely the data object it is writing. This way, the middleware can distinguish the instance being written and decide, for example to keep only the most current value. The performance and fault-tolerance requirements of real-time applications make centralized approaches impractical. Hence, it is not reasonable to expect that the "true value" of each data-object will "live" in a single computer. This implies that (a) there must be a global way to identify the instances of data objects, and (b) ownership QoS must be carefully defined to not force a "centralized" implementation. Topics already provide a network-wide addressing scheme. However, for applications with large numbers of data objects it is not practical (due to the overhead introduced by topic propagation) to introduce a different topic for each data-object instance.

To reduce the number of topics, the OMG DDS uses the combination of a ***Topic*** object introduced in Section 1 and a ***key*** to uniquely identify data-object instances. The representation and format of the key depends on the data type. However, since a ***Topic*** is bound to a unique type, the service can always interpret the key properly given the ***Topic*** and the value of a data object.

The combination of a fixed-type ***Topic*** and a ***key*** is sensible for data-centric systems because the ***Topic*** represents either a unique data object (e.g. a temperature sensor) in the case where there are no keys, or a set or related data-objects that are treated uniformly (e.g. track information of aircraft as generated by a radar system), where each individual aircraft can be distinguished by a key. The DDS delegates to the type the interpretation of the key so that it is possible for the key to be a single value within the data-object (e.g. a serial number field) or a combination of fields (e.g. airline-name and flight-number).

This use of a key is unique to data-centric systems [2, 4] and is neither used in "enterprise" publish-subscribe systems [7, 6, 8, 11] nor in event-distribution systems [9].

3 Read and write access

Another distinguishing aspect of publish-subscribe systems is the interface used to read and write global data-objects (see Figure 2).

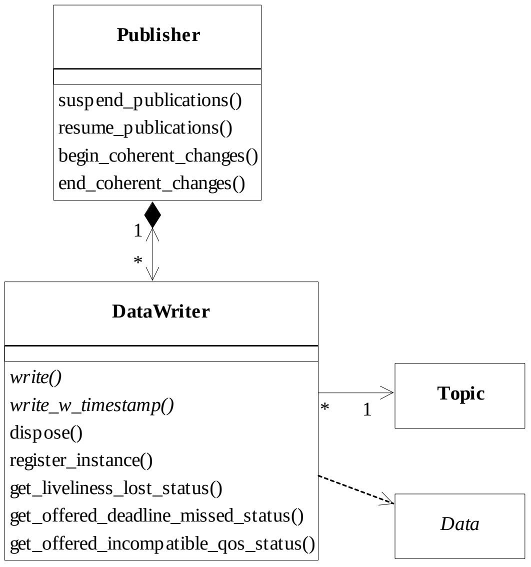

Figure 2 OMG Write interface

3.1 Publishing interface

The participant writes data using the ***write*** operation on the ***DataWriter***. The ***write*** operation takes as a parameter an object of the appropriate type (i.e. the type specified when defining the ***Topic*** associated with the ***DataWriter***).

The ***write*** operation just informs the middleware that there is a new value for the data object. It does not necessarily cause any immediate network communications. The actual generation of messages is controlled by the ***Publisher*** and the QoS. This is key because in a real-time system it may be important that the actual transport write is performed by a separate, lower-priority thread. This can be configured via QoS.

The ***dispose*** operation also takes a data-instance as a parameter and requests the middleware to delete that instance of the data (identified by the key). The semantics of deletion is as follows: existing participants that have already received values for that Data instance will be made aware of the deletion by means of operations on the related ***DataReader***; participants that have not been previously informed of the existence of the ***Data*** instance will not see it at all.

The ***Publisher*** acts on behalf of one or several ***DataWriter*** objects that are related to it. When it is informed of a change to the data associated with one of its ***DataWriter*** objects, it is responsible for determining when to send, and actually sending, the data. This behavior is driven by the attached ***QoS.***

In addition, the operations ***suspend_publications*** and ***resume_publications*** provide a hint to the middleware that multiple data-objects within the ***Publisher*** are about to be written, and thus allow the middleware to use bandwidth more efficiently by batching the distribution of a set of writes. An implementation could disable the dissemination of messages and accumulate changes until ***resume_publications*** is called. This is only a hint. The middleware is free to send messages at any time and it must do so if, for example, other QoS, such as LATENCY_BUDGET would be violated.

The participant can also request that a set of changes be propagated in such a way that they are interpreted at the receivers' side as a consistent set of modifications[3]. For this, the Publisher[4] offers two operations, namely ***begin_coherent_changes*** to start a coherent set and ***end_coherent_changes*** to terminate it. These calls can be nested. The subject of coherent changes is explored more extensively in Section 4.

3.2 Subscribing interface

On the reading side there are two interaction styles: Listener-based and Wait-based.

3.2.1 Listener-based data access

In the Listener-based approach, the participant is notified of the appearance or change in value of data-objects by means of a Listener object that the user installs with the middleware. The ***Listener*** is an object that implements a specified interface. The DDS middleware informs the participant asynchronously by invoking the appropriate methods in the ***Listener***.

This approach is simple and efficient. Whenever a data value changes, or a new data object that matches a subscription appears, the middleware simply informs the participant. A possible disadvantage is that the access is performed in the context of a middleware thread; it may be more complex to weave the data read by this thread with the actions taken by other concurrent threads inside the same participant.

3.2.2 Wait-based data access

The wait-based approach provides a set of conditions that threads inside the participant can use to block while waiting for specific sets of changes. When any of the changes of interest occur, the thread is unblocked and can access the data directly in its own context. This interaction style is similar to the one obtained by using the UNIX select() call or the Win32 WaitForMultipleObjects() call. While this interaction style may be easier to handle in certain applications, the specification of the conditions and waits is more complex.

Regardless of whether the participant uses a listener-based or a wait based approach, the data is accessed by invoking the ***read*** or ***take*** operations on the ***DataReader***.

4 Semantics of state propagation

An important use case for data-centric publish subscribe systems is the propagation of state information. Here we use the word "state" in the classic meaning of system theory and state-machines. That is, the state of the system is the information needed to determine future responses without reference to the past history of inputs and outputs. The present state of the system, the present inputs and the sequence of future input interactions allow computation of all future states and output interactions. For example, the balance of a checking account is the state of the account system. Any sequence of transactions that resulted in that state will be treated the same in the future.

The very definition of state makes it clear that, other than for the purposes of logging, only the most current state matters.

In general, the state of a system is described by the combined values of a set of data objects that dynamic systems call the "state variables".

[3] This does not imply that the middleware has to encapsulate all the modifications in a single message; it only implies that the receiving application-process will behave as if it that were the case.

[4] This functionality is offered on a ***Publisher*** basis to respect a reasonable trade-off between usefulness and affordability. The counter-part (i.e. the ability to be made aware of a coherent set of modifications) has been defined on a ***Subscriber*** basis for the same reasons. Therefore coherence scope is restricted to the intersection between the publications of a ***Subscriber*** and the subscriptions of a ***Subscriber***.

State propagation is important because it provides a compact way for an application to model a remote system as well as allowing a late-joining participant to behave as if it had seen the complete history of the system.

Assume that the "state-variables" of a particular system are A, B, and C. Furthermore assume that these variables undergo changes in the following order: A1, B1, A2, B2, C1, A3. The corresponding sequence of states for the system: S1, S2, S3, S4, S5, S6 is given by the combined value of all state variables:

S1 = {A1}

S2 = {A1, B1}

S3 = {A2, B1}

S4 = {A2, B2}

S5 = {A2, B2, C1}

S6 = {A3, B2, C1}

Assume that a remote participant has subscribed to A, B, and C for the purposes of tracking the current state of the system. As it receives new values for A, B, and C, it reconstructs the state. Clearly (at least in some cases), it is not desirable that the remote participant infers states that never existed at the source. This would occur for example if the remote participant saw the value B2 before ever seeing the value A2, and thus reconstructed a state S = {A1, B2} that never existed at the source.

This section examines the proper sequences of states the remote participant should be allowed to see, and the support required from the DCPS middleware.

For data-centric systems, it seems reasonable to assume that, if so desired by the application, the DCPS service should ensure that:

(a) The states reconstructed by the subscribing participant should be restricted to states that actually existed in the publishing participant.

(b) The order in which the states are reconstructed on the subscribing participant should preserve the order in which the states happened in the publishing participant.

(c) If the state on the publishing side settles (i.e. does not change for "a while") the state seen by the subscribing participant should match that of the publishing participant.

These restrictions mean that the subscribing participant could see sequences such as:

S1,S2,S3,S4,S5,S6; or

S1, S3, S6; or

S5, S6; or

S6

But is should not see sequences such as

S3, S1, S6;	[violates order]
S1, S3	[does not settle on the last state S6]

4.1 Incoherent states

Sometimes multiple state variables must be updated together for the state to transition to the next coherent (or valid) state. Imagine for example that the latitude, longitude, velocity vector, and altitude of an aircraft are kept as three separate state variables A=(latitude, longitude), B=(velocity_vector) and C=(altitude)[5]. The DDS interface must provide the participant the means to update A, B, and C "atomically"[6] in the sense that the receiving participant should not be allowed to see a new value of A without simultaneously seeing the new value for B and C as well. Otherwise they may erroneously infer the aircraft is on a collision course.

This functionality is provided by the operations ***begin_coherent_changes*** and ***end_coherent_changes*** described in Section 3.

4.2 Presentation access units

In large systems, it may not be practical to model all the state variables as defining a single monolithic state. It may also not be practical to insist that all changes to state variables made by a participant are propagated in order without introducing delays. For this reason, the application may partition the state into separate independent units, each composed of several variables. DCPS refers to each of these units as an "access unit." DCPS offers several ways for the application to define the access units by means of grouping ***DataReader*** (***DataWriter***) objects under ***Publisher*** (***Subscriber***) objects and also by means of the PRESENTATION QoS policy.

5 Quality of service policies

The data-distribution service relies on the use of QoS (Quality of Service) to tailor the service to the application requirements. A QoS is actually a set of characteristics that drives a given behavior of the service. It is made of individual QoS policies (objects of type deriving from ***QoSPolicy***).

The description of all the QoS policies supported by the DCPS service is beyond the scope of this paper. Rather

[5] This admittedly contrived example is a simple illustration of the more general case.

[6] This does not mean DCPS supports transactions. The "atomicity" by DCPS to support coherent changes does not allow the change to be "aborted." Moreover, DCPS only serializes the changes of each Publisher (not across Publishers) and offers a weaker durability model.

we will only describe the general characteristics and provide some concrete examples.

A ***QoSPolicy*** can be set on all DCPSEntity objects. In many cases, for communications to occur properly, a ***QoSPolicy*** on the publisher side must be compatible with a corresponding policy on the subscriber side. For example, if a ***Subscriber*** requests to receive data reliably while the corresponding ***Publisher*** defines a best-effort policy, communication will not happen as requested. To address this issue and maintain the desirable decoupling of publication and subscription as much as possible, the specification for ***QoSPolicy*** follows the subscriber-requested, publisher-offered pattern. In this pattern, the subscriber side can specify an ordered list of "requested" values for a particular ***QoSPolicy*** in decreasing order of preference. The Publisher side specifies a set of "offered" values for that ***QoSPolicy***. The middleware will then pick the most-preferred value requested by the subscriber side that is offered by the publisher side, or may reject the establishment of communications between the two ***DCPSEntity*** objects if the QoS requested and offered cannot be reconciled.

The following table lists some of the supported ***QoSPolicy*** options.

DEADLINE Parameters: A duration "deadline_period"	***DataReader*** expects a new sample updating the value of each instance at least once every *deadline_period*. ***DataWriter*** indicates that the participant commits to write a new value for each instance managed by the ***DataWriter*** at least once every *deadline_period*.
LATENCY_BUDGET Parameters: A duration "delay_laxity"	Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing participants.
TIME_BASED_FILTER Parameters: A duration "minimum_separation"	Filter that allows a ***DataReader*** to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the ***DataReader*** does not want to receive more than one value each *minimum_separation*, regardless of how fast the changes occur.
CONTENT_BASED_FILTER A string "expression" and a sequence of strings "parameters"	[optional] Filter that allows a ***DataReader*** to filter the data received from a given ***Topic*** based on the contents of the data itself. Syntax of "expression" is like an SQL WHERE clause. Only the parameter part may be changed.
PRESENTATION An "access_scope": INSTANCE, TOPIC, GROUP And two booleans: "coherent_access" "ordered_access"	This policy affects the participant's ability to: (1) Specify and receive coherent updates. (2) See the relative order of updates. The "access_scope" determines the largest scope spanning the entities for which the order and coherency of updates can be preserved. The two booleans control whether "coherent access" and "ordered access" are supported within the "access_scope."
	INSTANCE scope. Scope spans only a single instance. Coherence and order apply to each instance separately.
	TOPIC scope. Scope spans to all instances within the same ***DataWriter*** (or ***DataReader***), but not across instances in different ***DataWriter*** (or ***DataReader***).
	[optional] GROUP scope. Scope spans to all instances belonging to ***DataWriter*** (or ***DataReader***) entities within the same ***Publisher*** (or ***Subscriber***).

In addition to the values in the table, the following QoS policies are also supported: DURABILITY, OWNERSHIP, LIVELINESS, PARTITION, RELIABILITY, and DESTINATION_ORDER.

6 Relation to other standards

The specifications most closely related to the DDS are the OMG Notification Service [9] and the High-Level Architecture (HLA) [4].

6.1 The OMG Notification Service

This service models information exchange as events flowing in a channel. Applications must attach to channels (which can be named and accessed from a name server) and then use Supplier and Consumer objects to produce and consume the events. Events are independent entities that can have optional fields that allow an application to filter for the events of interest. QoS settings on the channel and on each event allow the notification service to schedule the underlying communications (for instance batching events to save bandwidth) and re-arrange the order in which events are delivered (for example, according to priority). Events can have a data payload but there is no explicit concept that associates events to changes on the values of global data structures.

It would be possible for an application developer to use the Notification Service to propagate the changes to data structures and in this manner provide the functionality of the DDS. However, doing this would be significantly complex because the Notification Service does not have a concept of data objects or data-object instances nor does it have a concept of state coherence. The application designer would have to develop these services on top, adding significant complexity to the Notification Service.

Similarly, it is conceivable to build the DDS on top of the NS. This layered approach, however, is not likely to be as efficient as an implementation that directly maps the DDS API into the implementation middleware without serializing each change as an event.

6.2 The High-Level Architecture (HLA)

HLA, also known as the OMG Distributed Simulation Facility, is a standard from both IEEE and OMG. It describes a data-centric publish-subscribe facility and a data model. The OMG specification is an IDL-only specification and can be mapped on top of multiple transports. The specification address several of the requirements of data-centric publish subscribe: The application uses a publish-subscribe interface to interact with the middleware, it includes a data model and supports content-based subscriptions. However, the HLA model is quite specific to combat simulations; the data model supports a specialization hierarchy, but not an aggregation hierarchy. The set of types defined cannot evolve over time. Moreover, the data elements themselves are un-typed and un-marshaled (they are plain sequences of octets). HLA also offers no generic QoS facilities.

7 Conclusion

Many real-time applications have a requirement to model some of their communication patterns as a pure data-centric exchange where applications publish (supply or stream) “data” which is then available to the remote applications that are interested in it. These types of real-time applications can be found in C4I systems, industrial automation, distributed control and simulation, telecom equipment control, and network management. Of primary concern to these real-time applications is the efficient distribution of data with minimal overhead and the ability to control QoS properties that affect the predictability, overhead, and resources used. Distributed shared memory is a classic model that provides data-centric exchanges. However, this model is difficult to implement efficiently over the Internet.

Therefore, another model, the Data-Centric Publish-Subscribe (DCPS) model, has become popular in many real-time applications. While there are several commercial and in-house developments providing this type of facility, to date, there have been no general-purpose data-distribution standards. As a result, no common models directly support a data-centric system for information exchange.

The OMG Data-Distribution Service (DDS) is an attempt to solve this situation. This specification defines a model of a data-centric publish-subscribe system in terms of objects such as ***DomainParticipant***, ***DataWriter***, ***Publisher***, ***DataReader***, ***Subscriber***, and ***Topic***. The specification also defines the operations and QoS attributes each of these objects supports and the interfaces an application can use to be notified of changes to the data or wait for specific changes to occur.

Bibliography

1. Oki, B., Pfluegl, M., Siegel, A., Skeen, D., "The Information Bus -- An Architecture for Extensible Distributed Systems", SOSP14, pp 58-68, Dec, 1993.
2. M Boasson and E. de Jong. *Control System Software*. IEEE Transactions on Automatic Control, Vo. 38, No. 7, July 1993.
3. Françoise Fabret, H.-Arno Jacobesen, François Llirbat, João Pereira, Kenneth Ross, Dennis Shasha. *Filtering Algorithms and Implementation for very fast publish/subscribe systems*. SIGMOD Conference, Santa Barbara, CA. May, 2001.
4. HLA: Distributed Simulation Systems V1.1 Document **formal/2000-12-01** www.omg.org.
5. Gerardo Pardo-Castellote, Stan Schneider, Mark Hamilton, *NDDS: The Real-Time Publish-Subscribe Network*, Real-Time Innovations, Inc. White Paper http://www.rti.com, 1999.
6. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*. http://www.vitria.com, 1998.
7. TIBCO. TIB/Rendezvous White Paper. http://www.rv.tibco.com/, 1999. 6.
8. Talarian Corporation “Mission critical Interprocess Communications – an Introduction to Smartsockets,” whitepaper www.talarian.com.
9. OMG The CORBA Notification Service. Document orbos/98-11-01. http://www.omg.org.
10. OMG Data-Distribution Service for Real-Time Systems. http://www.omg.org/cgi-bin/doc?mars/2003-01-05
11. The Java Messaging Service (JMS) specification. http://java.sun.com/products/jms/.