A Data Distribution Services and Instant Messaging Integration Architecture

Javier Povedano Molina, Juan M. Lopez-Soler

December 15, 2008

Signal Theory, Telematics, and Communications Department
University of Granada.

Abstract

The Internet evolution over recent years has originated new needs of communication. The classic centralized server/client approach has become not adequate in some scenarios and therefore new paradigms have been designed to satisfy the new requirements. The data-centric publish/subscribe is one of these new paradigms.

Data Distribution Service (DDS) is a data-centric publish/subscribe standard proposed by the Object Management Group (OMG) for real-time communications. Applications developed under the publish/subscribe paradigm are distributed in nature. This involves a number of problems that need to be solved. For example, it is difficult to trace all the participants and their participation conditions. Added to that, in general the interfaces used when dealing with distributed applications are very complex, and generally not designed for non-experimented users.

Instant Messaging (IM) services have become very popular on the Internet. Almost every user on the net knows how to use these services. The ease of use and the simple interfaces provided are the main reasons for their success.

The present project proposes a new approach that merges both technologies (DDS and IM) into a single framework of operation. More precisely, the main purpose is to take advantage of the use of Instant Messaging interfaces for accessing Data Distribution Services environments. A special kind of instant messenger agents called bots are designed for being the interaction point or interface between the user and Data Distribution Services. For such goal, a semi-natural language is defined.
# Contents

1 Introduction  
1.1 Concepts, Definitions and Acronyms .......... 4  
1.2 Technologies Overview  
1.2.1 Data Distribution Service .......... 5  
1.2.2 Instant Messaging Services .......... 6  
1.3 Problem Description  
1.3.1 Objectives .......... 8  
1.3.2 Requirements .......... 8  

2 Proposed Architecture .......... 8  
2.1 Notification Module .......... 9  
2.2 DDSXML Module  
2.2.1 DDSXML Classes .......... 11  
2.2.2 DDSXML Document Type Declarations .......... 12  
2.2.3 Transformation .......... 18  
2.3 Discovery Module .......... 20  
2.4 Dynamic Type Module  
2.4.1 Serialization and Deserialization .......... 23  
2.4.2 Data Representation Mapping .......... 25  
2.5 Interpreter Module .......... 26  

3 Grammar Description .......... 27  
3.1 Notification grammar .......... 27  
3.2 Status access operations .......... 28  
3.3 Setting/getting operations .......... 29  
3.4 Retrieve operations .......... 30  

4 Integration with Instant Messenger client .......... 30  
4.1 pidgin/purple .......... 30  
4.1.1 Protocol Plugins .......... 31  
4.2 DDS Accounts .......... 32  
4.3 DDS Bots .......... 34  

5 Presence and Status .......... 35  
5.1 Visual status information .......... 35  
5.2 Textual status information .......... 36  

6 Application Deployment .......... 37  

7 Examples and Results .......... 38  
7.1 Examples .......... 38  
7.1.1 Connection to a dds domain with an account .......... 38
7.1.2 Set a QoS via a profile .......................... 39
7.1.3 Set notification filtering ............................ 39
7.1.4 Asking about entity status .......................... 40
7.2 Results .................................................. 41
1 Introduction

The main goal of this work is to access Data Distribution Service information with an Instant Messaging approach, in this way we simplify and ease the access to Data Distribution Services information and entities.

To explain the proposed design and its implementation the present document is organized as follows:

- The first section describes the most important concepts used in this project. In addition, the project itself is also described.
- The section 2 explains the proposed architecture
- In section 3 we define the grammar used for providing interaction between the user and DDS entities
- Section 4 explains how to integrate the two independent paradigms (DDS and IM)
- Section 5 considers two additional issues: presence information management and status control
- Section 6 provides some use cases and examples
- And finally, the obtained results and conclusions are presented in Section 7

1.1 Concepts, Definitions and Acronyms

This section is aimed to describe the main terms and concepts behind this project. They are related mostly to Data Distribution Service and Instant Messaging Services.

Data Distribution Service (DDS) the publish/subscribe middleware that standardizes a data-centric publish-subscribe programming model for distributed real-time systems.

Domain an isolation mechanism for DDS entities. Applications use domains to exchange data with entities in other domains.

Domain Participant Entry point for sending and receiving information in a specific domain; it represents the participation of an application in one DDS Domain. Furthermore, it acts as a factory for the creation of DDS Publishers, Subscribers, Topics, MultiTopics and ContentFilteredTopics.

Publisher the object responsible for the actual dissemination of publications.

Subscriber the object responsible for the actual reception of the data resulting from its subscriptions.

Topic data exchanged in publications and subscriptions between peers. At a high level description, a topic consists on the name of the publication and a description of the data type used in the publication.
1.2 Technologies Overview

This section is devoted to briefly review the main technologies involved on the project. In particular, Data Distribution Services and Instant Messaging concepts are considered here.

1.2.1 Data Distribution Service

\textit{Data Distribution Service}\cite{1, 2} is a middleware standard specification that provides an Application Programming Interface (API) for developing distributed applications with real-time requirements. It adopts the publish/subscribe paradigm. Initially proposed by \textit{Real-Time Innovations} and \textit{Thales Group} companies, it has become the dominant standard technology since the \textit{DDS specifications} were approved by the \textit{Object Management Group (OMG)}\cite{3} in 2004.

The publish/subscribe paradigm is an alternative to the classic server/client architecture. In this data-centric paradigm the nodes exchange data automatically according to their interests, instead of pulling the data explicitly from a centralized server. When a node participates in the DDS data-space, it announces the data it can provide and the data in which it is interested in as well.

\textit{Data Distribution Service} middleware has the responsibility to match the nodes that must exchange information according to their needs. The middleware it's not only responsible for matching the remote entities based on their interests, it also has the responsibility of satisfying a bunch of Quality of Service (QoS) requirements imposed by the peers that participates on a matching or subscription.

In figure 1 an example of \textit{Data Distribution Service} scheme has been depicted.
1.2 Technologies Overview

1.2.2 Instant Messaging Services

Instant messaging is an application that enables networked users to send and receive short text messages. It has become an attractive alternative for quick communication over the Internet replacing the email in some cases. More recently, it is also being considered as an alternative to SMS for mobile users. There are a lot of instant messaging frameworks like Windows Live Messenger[4], Yahoo Messenger[5] and Skype[6] that connect millions of people all over the world everyday.[7]

One of the most important features of the Instant Messaging services is the ease of use. This feature has enabled the communication between inexperienced users over the Internet. Instant Messaging services interfaces are so intuitive and easy to understand that the learning curve for novice users is extremely low.

Some companies have taken advantage of the popularity and ease of use of these approaches for providing innovative services. Recently, the inclusion of non-human users in the instant messenger service is becoming popular. The range of tasks performed by this non-human users is very wide including forecast reports to instant translations[8] or access to encyclopedia services as well. The non-human users appear in the instant messenger program, and acts as a human. It communicates with the user by exchanging natural language text messages.

In Figure 2 an example of an non-human user is shown. In this case, the user chats with an automatic translation machine service installed in Windows Live Messenger.
1.3 Problem Description

1.3.1 Introduction

This service is part of the Microsoft Research Machine Translation project [9]. The procedure to interact with this service is to chat with it as a regular user with the use of a semi-natural language: the user provides some sentences in some language, and the other part on the conversation answers with the translation to a preestablished language. The user acts like asking to some human friend about a translation of a phrase and the machine answers it in a natural language too, so the communication is very natural for the user.

As mentioned before, this kind of services are proliferating now on most of the instant messaging services. The reasons for this popularity are:

- The user doesn’t need to install new programs to access to new services
- It saves time to the users as they don’t have to lookup for an specific service over the internet. For example, user doesn’t need to search about translation services on Google if you have already a translation buddy in the contact list of your instant messenger service.
- The user doesn’t expend time on learning how to run a program. It is as easy as to establish an informal conversation with a friend.

1.3 Problem Description

Data Distribution Service is a complex middleware for establishing data exchange between nodes in a distributed environment. Like most of the distributed systems, the complexity of Data Distribution Service makes it difficult to manage and monitor it.
The idea that motivates this project is to make a DDS based distributed system simpler by simplifying the interfaces to access to Data Distribution Service. The approach chosen for this purpose is to build a natural language based system to access to the high-level operations that DDS middleware allows.

The ease of use of Instant Messenger services by non expert uses makes it as an alternative for developing new interfaces to specialized services. As referred in section 1.2.2, a lot of services have been integrated into a instant messagery infrastructure by the use of natural language interaction. This kind of services are called bots.

The project presented in this document tries to take advantage of these bots to access Data Distribution Service without the need of complex interfaces or programs; the interaction is done as regular conversations but in this case with specialized bots.

1.3.1 Objetives

According to previous section it is easy to see that the objectives of this project are:

- To provide an alternative way of interacting with Data Distribution Service by using a defined natural language
- To provide a general purpose methods for accessing to Data Distribution Service without the need of coding

1.3.2 Requirements

**Automatic Discovery** The DDS entities in the scenario are discovered by the Instant Messenger client using the BuiltinTopics provided by RTI DDS[10] (BuiltinTopics is not covered by the OMG standard). This information is stored internally as XML trees for easy processing and formatting for user queries.

**JSON Support** All the information provided by RTI DDS is transformed into JSON[11] format. Other output formats can be easily added.

**Multiple Domains Support** As the instant messaging application supports multiple accounts to be connected at the same time, we use this feature for supporting multiple DDS Domains.

**New Types Support** The information about typecodes provided in the discovery process is used to support new topic types without re-compiling.

**Multiple Data Access Schemes** The application can retrieve the information from DataReaders synchronously or triggered by user request.

2 Proposed Architecture

In order to achieve all the requirements mentioned on section 1.3.2, a modular architecture has been proposed. The designed modular architecture tries to isolate the two main parts of the project:
2.1 Notification Module

When a Data Distribution Service application is running, usually a lot of threads are managed concurrently. Each of these threads is running a specific low level task (polling DataReader, running Discovery Protocol,...). When a specific event occurs on one of these threads, the Data Distribution Service notifies the application about this event using an specific API callback. In this sense, the Notification Module is a bridge between

- User Interaction: this module controls how the user and the bot interact. It includes the information exchange format definition and the graphical user interface (GUI) control.
- DDS Interaction: this part is aimed to deal with the DDS layer. It acquires information about discovered entities, samples and the API access by user request.

The architecture is depicted on figure 3. The most important modules are shown:

- Notification
- Discovery
- Dynamic Type
- DDSXML
- DDSJSON
- Interpreter

A detailed description of these modules is done in next sections.
the DDS event notification using entity callbacks and the instant messenger client. The Notification Module consists of two main parts:

**Notification object:** A DDS event is encapsulated into a Notification object that contains some information. It includes the type of the notification and a message associated to the notification.

**Notification Manager:** The notification manager delivers the notifications received from the DDS middleware to the appropriated bot conversation.

When an event occurs in a *Data Distribution Service* thread, the Notification Manager receives this notification (the Notification Manager is a thread-safe object so it can receive notifications from multiple threads at the same time). When there are available notifications, they are delivered to the user by adopting the instant messenger paradigm: all the events related to an specific topic are written in the established conversation between the user and a buddy that is created with the same topic name.

The notification manager dispatcher also includes a simple filter system. This mechanism allows to deliver only certain kind of notifications, according to the user requirements. The filter can be configured in a per-topic basis: only a selected subset (configured by the user) of notifications will be delivered, the rest will be dropped. The set of available notifications are:

**Discovery Notifications** To notify the discovery of remote entities event by the local DDS middleware

- **Publication Discovered** A remote publication of a certain topic was discovered.
- **Subscription Discovered** A remote subscription of a certain topic was discovered.

**Sample Notifications** This type of notifications are related to topic sample matters. Three kind of notifications are considered:

- **Sample Arrival** A sample has arrived on the local Data Reader matched.
- **Sample Lost** A sample has been lost somewhere in the path from the *Data Reader* to the *Data Writer*. It is only effective on publications that were subscribed using the `reliable` QoS policy.
- **Sample Rejected** A sample was rejected by the *Data Reader* because of any QoS negotiated policy couldn’t be verified. An example of sample rejection is when *lifespan* (configured by QoS settings) of a topic sample has expired according to the negotiated QoS policies.

**Liveliness Notifications** These are related to the liveliness status of *DDS Entities*. Signaling information is sent across DDS to determine whether an entity is alive. *Data Readers* should be aware of the liveliness of remote *Data Writers*.

- **Liveliness Changed** This notification is sent when a remote *Data Writer* is not alive for a local *Data Reader*. There might be many reasons for that, for instance remote application closed, connection failure, etc.
Liveliness Lost This event occurs in a Data Writer that wasn’t able to send heartbeat information for remote Data Readers, so this Data Reader considers this Data Writer as not alive.

Deadline Notifications These are caused by the deadline expiration of a topic sample. Deadline refers to the maximum time elapsed between topic sample updates and it is negotiated on the publication/subscription matching procedure.

Offered Deadline A Data Writer wasn’t able to respect the deadline period established with a remote Data Reader. This notification takes place in the writer side.

Requested Deadline A Data Writer wasn’t able to respect the deadline period established with a remote Data Reader. This notification takes place in the reader side.

As we can see, the notification categories are very similar to the Data Distribution Service Status. Each time a status changes in DDS, a notification is sent.

As it was described above, the Notification Manager proceeds like an event notification router that collects all the status changes from the DDS middleware and sends it to the appropriate bot. In other words, it sends this status information to the bot that handles the topic that originates the event. In some cases these notifications might be dropped by the Notification Manager if the user specifies the appropriated notification filter.

2.2 DDSXML Module

All the information that can be retrieved from the Data Distribution Service using the API is returned in some native structures. These structures are not human-readable and depends directly from the API. The basic idea for designing this module is to make all these structures human-readable and easy to transform in other different formats.

2.2.1 DDSXML Classes

DDSXMLQos This submodule converts Data Distribution Service QoS policies into XML format. Currently only the most important QoS policies are supported for this conversion. These policies are the ones that are exchanged in the discovery process, and are the ones that make the publication and subscription compatibility checking possible.

DDSXMLStatus This submodule converts Data Distribution Service status information into XML format. The Data Distribution Service standard defines a set of status in which the DDS Entities can be. These status contain information related to the topic samples being published on a certain domain and some extra information about extra information like QoS violations. For example, DDS Status holds information about the number of times that a deadline had been missed for a certain topic.
2.2 DDSXML Module

**DDSXMLEntities** The *builtin topics* are *special* topics that contain relevant information necessary in the discovery process. They are handled by the *Data Distribution Service* without user intervention. This submodule captures and converts the *builtin topic* data information into an internal XML format. The current *builtin topics* that *Data Distribution Service* supports are:

- **ParticipantBuiltinTopicData** Information about remote participants in a certain *domain*.
- **PublicationBuiltinTopicData** Information about publication endpoints (remote *DataWriters*) and their QoS offerings
- **SubscriptionBuiltinTopicData** Information about subscription endpoints (remote *DataReaders*) and their QoS requirements
- **TopicBuiltinTopicData** Information about topics being published on *Data Distribution Service*

### 2.2.2 DDSXML Document Type Declarations

When dealing with XML documents and trees, it’s usual to define the allowed tags, and the content and hierarchic relationships between the tags as well. In this section we describe the XML contents used in our architecture. We use *Document Type Definition (DTD)*[12] as a form of definition of valid XML documents. It is not as powerful as other XML Schema[13], however given its simplicity DTD is adopted given that it provides the required functionality.

#### Listing 1: DTD of DDSXMLEntities

```xml
<ELEMENT participant (key,entity_name,rtps_protocol_version,
    product_version,default_unicast_locators,property_qos)>
<ELEMENT publication (key,topic_name,type_name,participant_key,
    lifespan,deadline,partition,latency_budget,liveliness,
    durability,reliability,property_qos,typecode)>
<ELEMENT sample (struct)>
<ELEMENT subscription (key,topic_name,type_name,participant_key,
    deadline,partition,latency_budget,liveliness,durability,
    reliability,presentation,destination_order,time_based_filter,
    unicast_locators,multicast_locators,property_qos)>
```

#### Listing 2: DTD of DDSXMLQos

```xml
<ELEMENT entity_name (name)>
<ELEMENT rtps_protocol_version (major,minor)>
<ELEMENT product_version (major,minor,release,revision)>
<ELEMENT major (#PCDATA)>
<ELEMENT minor (#PCDATA)>
<ELEMENT release (#PCDATA)>
<ELEMENT revision (#PCDATA)>
```
2.2 DDSXML Module

Listing 3: DTD of DDSXMLDataReaderStatus

```xml
<ELEMENT dataraster_status (liveliness_changed_status, sample_lost_status, sample_rejected_status, requested_deadline_status, subscription_matched_status, dataraster_protocol_status)>

<ELEMENT liveliness_changed_status (alive_count, alive_count_change, not_alive_count, not_alive_count_change)>
```
<ELEMENT sample_lost_status (total_count, total_count_change)>
<ELEMENT total_count #CDATA>
<ELEMENT total_count_change #CDATA>

<ELEMENT requested_deadline_status (total_count, total_count_change)>

<ELEMENT subscription_matched_status (total_count, total_count_change, current_count, current_count_peak, current_count_change)>
<ELEMENT current_count #CDATA>
<ELEMENT current_count_peak #CDATA>
<ELEMENT current_count_change #CDATA>

<ELEMENT datareader_protocol_status (received_sample_count, received_sample_count_change, received_sample_bytes, received_sample_bytes_change, duplicate_sample_count, duplicate_sample_count_change, duplicate_sample_bytes, duplicate_sample_bytes_change, filtered_sample_count, filtered_sample_count_change, filtered_sample_bytes, filtered_sample_bytes_change, received_heartbeat_count, received_heartbeat_count_change, received_heartbeat_bytes, received_heartbeat_bytes_change, sent_ack_count, sent_ack_count_change, sent_ack_bytes, sent_ack_bytes_change, sent_nack_count, sent_nack_count_change, sent_nack_bytes, sent_nack_bytes_change, received_gap_count, received_gap_count_change, received_gap_bytes,}

Javier Povedano Molina 14 Universidad de Granada
2.2 DDSXML Module 2 PROPOSED ARCHITECTURE

Listing 4: DTD of DDSXMLDataWriterStatus
<!ELEMENT datawriter_status (liveliness_lost_status, offered_incompatible_qos_status, offered_deadline_status, 

received_gap_bytes_change,
rejected_sample_count,
rejected_sample_count_change)>

<!ELEMENT received_sample_count #CDATA>
<!ELEMENT received_sample_count_change #CDATA>
<!ELEMENT received_sample_bytes #CDATA>
<!ELEMENT received_sample_bytes_change #CDATA>
<!ELEMENT duplicated_sample_count #CDATA>
<!ELEMENT duplicated_sample_count_change #CDATA>
<!ELEMENT duplicated_sample_bytes #CDATA>
<!ELEMENT duplicated_sample_bytes_change #CDATA>
<!ELEMENT filtered_sample_count #CDATA>
<!ELEMENT filtered_sample_count_change #CDATA>
<!ELEMENT filtered_sample_bytes #CDATA>
<!ELEMENT filtered_sample_bytes_change #CDATA>

<!ELEMENT received_heartbeat_count #CDATA>
<!ELEMENT received_heartbeat_count_change #CDATA>
<!ELEMENT received_heartbeat_bytes #CDATA>
<!ELEMENT received_heartbeat_bytes_change #CDATA>

<!ELEMENT sent_ack_count #CDATA>
<!ELEMENT sent_ack_count_change #CDATA>
<!ELEMENT sent_ack_bytes #CDATA>
<!ELEMENT sent_ack_bytes_change #CDATA>

<!ELEMENT sent_nack_count #CDATA>
<!ELEMENT sent_nack_count_change #CDATA>
<!ELEMENT sent_nack_bytes #CDATA>
<!ELEMENT sent_nack_bytes_change #CDATA>

<!ELEMENT received_gap_count #CDATA>
<!ELEMENT received_gap_count_change #CDATA>
<!ELEMENT received_gap_bytes #CDATA>
<!ELEMENT received_gap_bytes_change #CDATA>

<!ELEMENT rejected_sample_count #CDATA>
<!ELEMENT rejected_sample_count_change #CDATA>
publication_matched_status,
datareader_protocol_status)
>
<ELEMENT liveliness_lost_status (alive_count,
alive_count_change,
not_alive_count,
not_alive_count_change)>

<ELEMENT offered_incompatible_qos_status (total_count,
total_count_change)>
<ELEMENT total_count #(CDATA)>
<ELEMENT total_count_change #(CDATA)>

<ELEMENT offered_deadline_status (total_count,
total_count_change)>

<ELEMENT publication_matched_status (total_count,
total_count_change,
current_count,
current_count_peak,
current_count_change)>
<ELEMENT current_count #(CDATA)>
<ELEMENT current_count_peak #(CDATA)>
<ELEMENT current_count_change #(CDATA)>

<ELEMENT datwriter_protocol_status (pushed_sample_count,
pushed_sample_count_change,
pushed_sample_bytes,
pushed_sample_bytes_change,
pulled_sample_count,
pulled_sample_count_change,
pulled_sample_bytes,
pulled_sample_bytes_change,
filtered_sample_count,
filtered_sample_count_change,
filtered_sample_bytes,
filtered_sample_bytes_change,
sent_heartbeat_count,
sent_heartbeat_count_change,
sent_heartbeat_bytes,
sent_heartbeat_bytes_change,
received_ack_count,
received_ack_count_change,
2.2 DDSXML Module

received_ack_bytes,
received_ack_bytes_change,
received_nack_count,
received_nack_count_change,
received_nack_bytes,
received_nack_bytes_change,
sent_gap_count,
sent_gap_count_change,
sent_gap_bytes,
sent_gap_bytes_change,
rejected_sample_count,
rejected_sample_count_change)

<!ELEMENT pushed_sample_count #CDATA>
<!ELEMENT pushed_sample_count_change #CDATA>
<!ELEMENT pushed_sample_bytes #CDATA>
<!ELEMENT pushed_sample_bytes_change #CDATA>
<!ELEMENT pulled_sample_count #CDATA>
<!ELEMENT pulled_sample_count_change #CDATA>
<!ELEMENT pulled_sample_bytes #CDATA>
<!ELEMENT pulled_sample_bytes_change #CDATA>
<!ELEMENT filtered_sample_count #CDATA>
<!ELEMENT filtered_sample_count_change #CDATA>
<!ELEMENT filtered_sample_bytes #CDATA>
<!ELEMENT filtered_sample_bytes_change #CDATA>

<!ELEMENT sent_heartbeat_count #CDATA>
<!ELEMENT sent_heartbeat_count_change #CDATA>
<!ELEMENT sent_heartbeat_bytes #CDATA>
<!ELEMENT sent_heartbeat_bytes_change #CDATA>

<!ELEMENT received_ack_count #CDATA>
<!ELEMENT received_ack_count_change #CDATA>
<!ELEMENT received_ack_bytes #CDATA>
<!ELEMENT received_ack_bytes_change #CDATA>

<!ELEMENT received_nack_count #CDATA>
<!ELEMENT received_nack_count_change #CDATA>
<!ELEMENT received_nack_bytes #CDATA>
<!ELEMENT received_nack_bytes_change #CDATA>

<!ELEMENT sent_gap_count #CDATA>
<!ELEMENT sent_gap_count_change #CDATA>
<!ELEMENT sent_gap_bytes #CDATA>
2.2 DDSXML Module

2.2.2 DDSXML Module 2 PROPOSED ARCHITECTURE

<table>
<thead>
<tr>
<th>XML format</th>
<th>Natural language format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;deadline&gt;</code></td>
<td>The deadline duration is 1 second.</td>
</tr>
<tr>
<td><code>&lt;duration&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;sec&gt;</code>1 <code>&lt;/sec&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;nanosec&gt;</code>0 <code>&lt;/nanosec&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;duration&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;deadline&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Transformation scheme example

```xml
<ELEMENT sent_gap_bytes_change #(CDATA)>

<ELEMENT rejected_sample_count #(CDATA)>
<ELEMENT rejected_sample_count_change #(CDATA)>
```

2.2.3 Transformation

Once the data is acquired in XML from the DDS middleware, this information should be transformed into a human-readable format. XML is a format that can be easily understood by the user and has the semantics necessary for our purposes. However, XML format itself contains so much textual overhead, decreasing legibility.

One of the main advantages of XML compared to other data representation formats is the ability of the format to be processed and transformed due to its hierarchical structure. When dealing with XML documents it is usual that documents have to be transformed to another XML document or format. For this goal, XML provides a very powerful tool to make the required transformations: XSLT\[14\].

XSLT is a XML subset aimed to transform XML documents using a set of defined templates. When processing an XML document against a XSLT stylesheet, every node in the XML document is matched against the templates defined on the XSLT document. If a template is matched then the transformation defined by the template is applied.

In our architecture, we will have some data in XML format relative to DDS entities or status. The XML data exhibit high hierarchical structure and big textual overhead. For example, for each opening tag in the XML file, there will be a matching closing tag.

In a conventional XML to natural language transformation, the followed procedure is very simple. Usually, we have some XML information acquired from Data Distribution Service like a discovered entity or an entity status. This information maintains a hierarchical structure that represents the relationship between the members that conform the XML document. Each of this members that refers to an unique concept (for example a QoS policy) is transformed into a given sentence in natural language that summarizes the information contained in the XML subtree that refers to the concept. The semantic of the concept is taken into an account when summarizing the content. An example is shown in figure 1 in which the deadline QoS policy has been summarized into a simple phrase, removing no relevant information (such as the 0 nanoseconds duration XML subtree) in
2.2 DDSXML Module

PROPOSED ARCHITECTURE

<table>
<thead>
<tr>
<th>XML format</th>
<th>JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sample&gt;</code></td>
<td>“foo” : {</td>
</tr>
<tr>
<td><code>&lt;struct name=&quot;foo&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;member name=&quot;bar&quot; type=&quot;short&quot;&gt;1&lt;/member&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;member name=&quot;woo&quot; type=&quot;string&quot;&gt;hello&lt;/member&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;member name=&quot;zoo&quot; type=&quot;boolean&quot;&gt;true&lt;/member&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/struct&gt;</code></td>
<td>}</td>
</tr>
<tr>
<td><code>&lt;/sample&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Transformation scheme example for topic samples

the final phrase.

There is an special case that is worth to explain further. It consists of the transformation of the data (topic samples) provided by DDS. When we transform a XML QoS policy into natural language, we have information about the semantic of the XML tree that defines the policy. We already know the components that define the QoS policy and their meaning. In the case of a deadline policy, for example we know that this policy only refers to a duration, so we can translate it in a simple sentence that says how long is a deadline. The textual information is more concise because we have semantic information. On the other hand, topics samples are very generic information: they can represent almost any kind of data. There is no semantic information about how to translate the XML topic sample in a sentence that summarizes the content.

To solve the problem of generic topic samples transformation into readable sentences, a lightweight data representation format has been chosen. This format, called JSON[11] (JavaScript Object Notation) easily allows the formal representation of complex objects. It is based in a pair name-value representation of data. With JSON we can represent the most important data types.

There is no unique mapping between JSON and XML, and therefore this mapping is not standardized yet. A lot of solutions to this issue have been proposed [15, 16], however none of them has been universally adopted. Some of them make use of XML namespaces for the mapping. This approach can be complex. For our architecture, we propose a simple mapping of XML topic samples to JSON format. The mapping is very straightforward and can be implemented in XSLT. This is a special case that is treated by a specific sample template in a XSLT stylesheet. The rules of XML to JSON representation are:

- a `<struct>` element tag is transformed into a JSON object
- each `<member>` of a struct becomes a JSON member pair name-value set
  - the pair name part comes from the name attribute of the `<member>` element
  - the value comes from the content of the `<member>` element
- complex data types such as list, sequences and arrays are transformed using the bracketed notation for representing list of elements in JSON (’[]’).

Javier Povedano Molina

19

Universidad de Granada
With this in mind, we can see a transformation example from XML topic sample representation to JSON data representation format in table 2.

2.3 Discovery Module

One of the most important differences of the RTI implementation of Data Distribution Service standard from their competitors is the auto-discovery of remote entities (publications and subscriptions). This discovery process is done in a transparent way, hiding the details of the process to the user.

The discovery process uses a protocol called SDP (Simple Discovery Protocol) that is built using one special kind of topics, called BuiltinTopics. These topics are spread among all the participants of a certain domain and proceed in two steps:

1. Discovering the Domain Participants

2. Discovering the remote endpoints associated to a remote publication/subscription (that is DataReaders and DataWriters)

The presence of Domain Participants is announced by publishing a ParticipantBuiltinTopic with participant information. Once the participants have discovered each other, the endpoints and topics present in a domain publish information about them using three kind of topics: TopicBuiltinTopic, PublicationBuiltinTopic, SubscriptionBuiltinTopic.

Although the discovery is hidden to the user as mentioned before, the user can access to this information accessing to the BuiltinTopics. In our case, it is necessary to access to the discovery information as we want to be notified about every publication and subscription present in a domain. Once we have the information about all the publications and subscriptions, maybe we want to establish a matching with this remote entities in order to exchange topic samples with them.

The present module is used to access the information exchanged in the auto-discovery process. This module install appropriate listeners in the Builtin Topic DataReaders and notifies the instant messenger client when an event related with a remote entity occurs. Then the received information about the remote entity is used to update the client, and to notify the user if necessary.

The task performed by this module are:

- Tell the Notification Manager (see 2.1) when a Domain Participant, publication or subscription is discovered.

- Get the relevant data about remote publications necessary to subscribe them and store it. This information includes topic name, topic type name and type code (for serializing/deserializing it properly).

In figure 4
2.3 Discovery Module

Figure 4: BuiltinTopics
2.4 Dynamic Type Module

Each topic in *Data Distribution Service* has an associated data type. When publishing or subscribing a topic sample in *Data Distribution Service*, the middleware is responsible for convert the sample from host native representation, into a common network representation used in the wire-protocol. On this process, some transformation such as endianness and word length conversions are performed. This process is called serialization when performed in host—network direction and deserialization when its done in network—host direction.

Usually, the topic types are known in compile time, so the serialization and deserialization methods are hardcoded into the software to provide very low overhead on (de)serialization process. In our case we want to access and subscribe to topics without prior knowledge about an specific data type. In other words, the topic type is only know for our application in execution time.

The current module provide mechanisms to deal with topics that have been discovered in dinamically. The client doesn’t know anything about the topic types before discovery, and all the topic type information is retrieved from discovery information.

These are the main tasks of this module:

- Subscribing to a discovered data type
- Serialize topic samples from intermediate language to DDS wire-protocol format
- Deserialize topic samples from DDS wire-protocol format into an intermediate language like XML
- Access to topic samples received by a *DataReader*

All the information to achieve this tasks it’s acquired by the Discovery module (sec 2.3). Once the Discovery module has the information relative to remote publications/subscriptions, the Dynamic Type module stores this information in order to perform the task assigned to this module.

This module contains a generic representation of a generic data type for a topic. Three elements are necessary to conform a *Dynamic Type* based topic: the name of the topic, the type name and the internal structure of it (*typeid*). As mentioned before all this information is colected by the Discovery module.

This module allows access to the topic samples in two modes:

**Synchronous** the user is notified about a new sample when it arrives. A *listener* installed in a local *DataReader* makes it possible. The callback installed in the *listener* uses the *Notification module* (sec 2.1) to inform the user about the sample arrival and content.

**Asynchronous** the user is able to access the topic samples on-demand. The user access to a topic sample by an explicit petition that is interpreted by the *Interpreter* module (which will be explained in sec. 2.5).
2.4 Dynamic Type Module

2.4.1 Serialization and Deserialization

Both processes, serialization and deserialization are very similar in concept: convert from one format of data representation into another different data representation. In our case we take the following formats into account:

Common Data Representation (CDR) Data Distribution Service represents the data in this format when publishing topics and receives the information in this formats when receiving topic samples on subscription. CDR format is standarized and defines some important information:

- Size of primitive data types such as short, long, boolean, ...
- Endianness of multiple bytes data types

On the Data Distribution Service, when a topic sample is delivered to a DataReader, it remains stored on it until one of this conditions is satisfied:

- The data sample is explicitely removed from the DataReader
- The DDS QoS constraints make one data sample non valid and it is removed automatically by the middleware. One example is the History QoS policy that limit the number of old samples that should remain stored.

Figure 5: Synchronous and asynchronous access
2.4 Dynamic Type Module

(a) Serialization  
(b) Deserialization

Figure 6: Serialization and Deserialization processes

- Packing format describing composites or aggregated data types such as arrays, sequences, struct, ...

JavaScript Object Notation (JSON) Is a lightweight data representation format, with very low overhead and very descriptive. It is very powerful and supports the most important primitive data types (integer, float, boolean, char, string) and some composite data types such as objects, arrays and lists. The main advantage is that is human-readable and it is very intuitive.

eXtensible Markup Language (XML) It is the most popular format on the internet and in many applications. Conceptually it is very simple and its hierarchical structure allows to represent many different kinds of data. As its name depicts, it is based on marks. The main advantage is that is human-readable and easy to transform in other formats (sec. 2.2).

In Data Distribution Service, when samples are published, the data put into the wire is not self-descriptive. An user cannot deserialize a topic sample only with the information of the topic sample. Data is published on the wire using CDR representation, and it only contains a stream of bytes without any extra information related with the stream structure. The main reason for that is that DDS minimizes the overhead of data being sent. This information about the CDR stream format is sent only once by DDS on the discovery process.

Another alternative is to hardcode the serialization and deserialization code explicitly in both publisher and subscriber side, but it is only useful in scenarios with topic data types that do not change or are statically defined. That case doesn’t match our application, that is supposed to publish and subscribe to types discovered dynamically.

In order to serialize and deserialize CDR Streams of a specific topic, this module makes use of the typecode that this topic has associated according to its type name. This information is discovered and acquired by the Discovery module. Once this information is available and there is a topic sample, the CDR Stream is serialized/deserialized according to the format that the typecode describes.
2.4 Dynamic Type Module

This module deserializes *CDR Streams* into XML representation which be transformed later into JSON representation. On the serialization process (publication), the data in JSON representation is directly transformed into *CDR Stream*.

### 2.4.2 Data Representation Mapping

As we mention in the section 2.4.1, there are three formats of data representation that come up in the data exchange between *Data Distribution Service*, the instant messenger client and the user. Those are CDR, XML and JSON. In this section we are going to depict how the mapping between types is done.

The mapping between this data representation formats is very straightforward. Some aspects should take into account when doing the mapping between data representation formats:

- **CDR format** needs a formal description of the data contained by the stream when serializing/deserializing, so *typecode* it’s needed to know how to serialize/deserialize the stream.

- Some metadata loss can occur on the conversion between data formats. For example:
  - CDR doesn’t include metadata about the format such as member names or data type information.
  - JSON doesn’t support information about data types. The data types are implicit.

- User only needs high level information about the types included inside the topics. In most cases the user only have to know to value of an integer, it doesn’t mind if the integer value is a *short* or a *long integer*.

One of the most important questions that the reader can make is ¿why not to use JSON on data and information retrieval from DDS instead of XML? JSON is more lightweight indeed and wouldn’t be difficult to extract data from *Data Distribution Service* in JSON format instead of XML. There are a couple of reasons not to choose this alternative:

<table>
<thead>
<tr>
<th>TypeCode</th>
<th>CDR</th>
<th>XML</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000002</td>
<td>struct a{&lt;br&gt;  octet b: &lt;br&gt;}</td>
<td><code>&lt;struct name=&quot;a&quot;&gt; &lt;member name=&quot;b&quot; type=&quot;octet&quot; a: 0 &lt;/member&gt; &lt;/struct&gt;</code></td>
<td><code>a : {b : 0}</code></td>
</tr>
</tbody>
</table>

Table 3: Data Mapping Representation
2.5 Interpreter Module

The interaction between the user and the Data Distribution Service is done with conversation between the user and a instant messenger bot. There is a bot for each of the topic discovered by DDS. Each of this bot is responsible of accessing to DDS according to the user instructions. Some of the tasks that should be performed by the bot are:

- Retrieve information about discovered entities
- Notify the user about relevant events related to a certain topic
- Retrieve information about the status of communications with remote publications/subscriptions
- ...

All this tasks can be specified by the user using a semi-natural language (see sec. 3). This module is responsible of translating the user requests into actions that interact directly with DDS.

The interpreter is responsible of this tasks:

1. Parse the user petition (made by the user in a conversation with a bot) and translate it into a specific action within the DDS API. If the petition cannot be translated into an action the user should be informed about a non well formed petition and suggest a petition format.

2. Commit the action

3. Return the information the results from step 2 in form of natural language feedback in a updating the user conversation with the bot.
3 Grammar Description

When creating bot programs, the most challenging part is the definition of the language that the bot should understand and translate into actions.\[17\]

The vocabulary and syntactic constructions that the bot should understand depends on the tasks that should be done by the bot. For example, the vocabulary for a conversational bot (that maintain trivial conversations without domain specific knowledge[18]), differs from the vocabulary and grammar of a bot that is aimed to maintain dialogues about music (very specific domain knowledge). There is many literature about incorporating domain specific knowledge to an instant messaging bot [17].

In our case, the language that the bot understands should be closely related to the Data Distribution Service knowledge domain.

The vocabulary related to Data Distribution Service can be classified in:

**DDS Entities** all the relevant entities of a DDS system should be recognised, such as *Participant, Publisher, Subscriber, DataReader, DataWriter,...*

**DDS Terms** other DDS terms (not entities) should be identified for the construction of valid statements. For example terms like *status, QoS, profile* and *library* should be recognised as they are concepts very common in a normal interaction with a Data Distribution Service.

**Actions** the grammar should be able to recognise actions like get or set that denotes the intention of the user to do this interactions with the DDS middleware.

**Primitive types** other primitive types that will be used as values on set operations. Examples of this primitives are integers and strings.

**Others** conversational connectors, specific format constructors and quantifiers should be also recognised. In this category we find examples like *all, nothing* and specific format constructors (like the JSON array ‘[]’ constructor).

Once the vocabulary has been defined, the next step is to define the grammar that the bot should recognise. The next part describe the grammar subsets the should be recognised, categorised by their functionality.

3.1 Notification grammar

<notify_stmt> ::= [notify|ignore] on sample [arrival|lost|reject]
| [notify|ignore] on liveliness change
| [notify|ignore] on [publication|subscription] discovered
| notify on all events
| ignore all [events]
| disable notifications

The notification grammar defines the Data Distribution Service events that should be notified to the user by the topic buddy. The events available to be notified to the user are:

Javier Povedano Molina

27

Universidad de Granada
3.2 Status access operations

Discovered Publication A remote publication of a certain topic was discovered in the Domain.

Discovered Subscription A remote subscription of a certain topic was discovered in the current Domain.

Sample Rejected A topic sample was rejected in the DataReader due some Qos constraint.

Sample Arrived A new topic sample was received in a DataReader and was accepted.

Sample Lost A sample of a certain topic was lost. Only takes effect on reliable topics.

Liveliness Change The liveliness of a remote endpoint (DataWriter) has changed. That means that a new DataWriter becomes alive or not alive.

Liveliness Lost The DataWriter wasn’t able to send an ‘alive’ signal to a remote DataReader, so the DataReader considers this DataWriter as not-alive. This

Deadline Missed The requested deadline period of a topic sample was missed for some reason.

Using this grammar we can set the notification mask of a certain topic. With this mask we can specify which notifications should be sent to the user within the bot, or which ones should be disposed.

3.2 Status access operations

<status_stmt> ::= local [datareader|dr] status
| local [datareader|dr] sample [lost|rejected]? status
| local [datareader|dr] liveliness [lost|changed]? status
| local [datareader|dr] subscription status
| local [datareader|dr] protocol status
| local [datareader|dr] deadline status

The Data Distribution Service entities have associated information that reflects the current status of the DDS Entity. These statuses. The available status are:

Sample Status: Stores information about the topic samples being published by other remote entities. It stores the total number of samples, the number of new samples since last check, the bytes received,... They can be categorized into two more specific classes:

Sample Lost Status: Stores information about the samples that were lost. Only on reliable publications.

Sample Rejected Status: Stores information about the locally rejected topic samples and the reason of rejection.

Liveliness Status This status stores information about alive/not-alive remote entities.
3.3 Setting/getting operations

Liveliness Lost The liveliness that the DDS_DataWriter has committed to through its DDS_LivelinessQosPolicy was not respected, thus DDS_DataReader entities will consider the DDS_DataWriter as no longer alive.

Liveliness Changed The liveliness of one or more DDS_DataWriter that were writing instances read through the DDS_DataReader has changed. Some DDS_DataWriter have become alive or not_alive.

Deadline Status The information about deadline status of topic samples

Offered Deadline The status about the deadline of the sent samples on DataWriter.
Requested Deadline The status about the deadline of topic samples on DataReader.

Incompatible QoS Status When matching a publication/subscription in DDS, first of all a compatibility checking between the remote endpoints is done. If there are incompatible Qos policies on both sides, the publication/subscription matching is aborted.

Offered Incompatible Qos The detection on an incompatible Qos on the publisher (DataWriter) side.
Requested Incompatible Qos The detection on an incompatible Qos on the subscriber (DataReader) side.

Protocol Status We can access some low-level statistics using this status. We can access for example, for the number of acknowledgments (ACK) sent to remote DataWriters

Subscription Matched Status Accessing this status we can get some information about the subscription matched.

Publication Matched Status Status information about a publication that has been matched.

Sometimes accessing to the whole set of status of a DataReader or DataWriter is not needed. The user might be interested only in a specific status of a local entity. For this reason the grammar allow accesses to the whole set of statuses at the same time, or only to a subset of them.

3.3 Setting/getting operations


<get_stmt> ::= get [participant|publisher|subscriber] default [library|profile]

This operations are related to the RTI Data Distribution Service Qos Profiles feature. With this operations we can manage and change some of the available QoS policies.

- Set/Get a default QoS profile

Javier Povedano Molina 29 Universidad de Granada
3.4 Retrieve operations

- Set/Get a default QoS library
- Set an entity QoS based on a profile name

Instead of setting the QoS profiles using specific statement for each QoS policy, we set the QoS policies using profiles. This reduces the complexity of the grammar and makes the QoS management easy.

3.4 Retrieve operations

\[
\text{<retr stmt>} ::= [\text{retrieve}\mid \text{retr}] \text{ last} \\
| [\text{retrieve}\mid \text{retr}] <\text{integer}> \\
| [\text{retrieve}\mid \text{retr}] \text{ all}
\]

This set of operations is aimed to access the samples received of a certain topic. Sometimes the user only need to access the last samples received at some point. With this set of operations we can access the last samples available on the local DataReader asynchronously. The number of samples that can be accessed with this operation depend on the QoS policies of the local DataReader. For example we can access only the last sample, all the samples available or only the \(N\) last samples.

4 Integration with Instant Messenger client

As a proof of concept an existing Instant Messenger client has been extended to support the functionality specified by this document.

Creating an instant messenger client from scratch supposes a lot of effort in design, implementation and testing. With the increasing popularity of the instant messaging services, the number of alternatives when choosing an instant messenger client has been increased too. A user is able to choose the messenger client that better fits their needs.

In the case of the DDS Instant Messenger Service presented here the most important features that have to be satisfied are:

- The client should be easy extended.
- It should be implemented using one of available programing language supported by the Data Distribution Service.

4.1 pidgin/purple

Pidgin/purple is the chosen client that better fits our purposes. pidgin/purple is a multiprotocol instant messenger client that can be easily extended.

The main features that are:

Multiple Protocols Support: pidgin/purple was conceived as a multiprotocol instant messenger client, so we can add DDS support as another supported protocol
4.1 pidgin/purple 4 INTEGRATION WITH INSTANT MESSENGER CLIENT

Figure 7: Pidgin Architecture

**Plugin Architecture:** pidgin/purple has a very powerful and extensible plugin architecture that makes easy the task of extend the client.

**User Interface Independent:** In pidgin/purple the instant messenger code and the user interface code are 100% independent, so we can develop extra core functionality that works with many different user interfaces.

This features are possible because of the internal architecture of pidgin/purple. This architecture is shown on figure 7.

While pidgin is the part of the client aimed to be an user interface to purple, purple is the module that holds the main functionality.

4.1.1 Protocol Plugins

As mentioned in previous section, *pidgin* has multiple protocol support. A lot of instant messenger protocols are built in by default on the *pidgin* client.

*Pidgin* provides an especial kind of plugin extensions to support new instant messenger protocols. These plugins define how to handle the incoming information and the incoming information in the new supported protocol. For example, a *pidgin IRC* protocol plugin have to define how to read the information from the server, how to parse and how to format and send the messages back to the server.
The work proposed here takes advantage of this feature and implements *Data Distribution Service* support for *pidgin* as a new *Instant Messaging* protocol. The mapping between a *Data Distribution Service* and a regular instant messenger service is not direct, so we need to have some ideas in mind:

- In *Data Distribution Service* data comes through subscriptions, so there is not an unique entry point for data. In most regular instant messenger services, the incoming data comes from a well-defined port connection. For this reason a special mechanism for handling incoming data should be defined.

- The data introduced by the user is not going to travel across the network. This data only needs to be translated into *Data Distribution Service* actions, so no outgoing message mechanisms should be provided, as the *Data Distribution Service* is responsible of the outgoing network data.

- Some mapping between *Data Distribution Service* concepts and *Instant Messenger* concepts should be done. In section 4.2 the case of accounting system is explained.

In the next sections two cases of the concept mapping between *Data Distribution Services* and *Instant Messenger services* are going to be explained: *accounting system* and *topic/buddy* association.

### 4.2 DDS Accounts

A instant messenger client make use of the concept of an account when connecting to an instant messenger service. This account has associated some things like an user name and a contact list. In the presented approach, an instant messenger account is also associated with a *DDS Domain*. When an user connects to the DDS Instant messenger service with an account, a *Domain Participant* on this *DDS Domain* is created associated to this account.

According to these rules there is a direct mapping between a DDSDomain Participant and a instant messenger account on the instant messenger client.

When creating the *Domain Participant* two more entities are created. Those are a Publisher and a Subscriber on this Domain Participant.

In figure 8 the internal structure of a DDS Account is depicted. Note that only one Domain Participant is created by instant messenger account and viceversa. Another fact is that only one publisher and subscriber are created per Domain Participant. Other entities like *Data Reader* and *Data Writers* are created on demand, so the number of this entities depend on the remote discovered publications and subscriptions.

A DDS account defines how the participation in a DDS domain is done. *Data Distribution Service* allows the user to define how this participation is done by setting some parameters. Those parameters are:

**DDS Domain** the domain where the application should lookup for publications and subscriptions.
4.2 DDS Accounts

Figure 8: DDS Account

Figure 9: DDS Account configuration
4.3 DDS Bots

Discovery Peers the discovery process in DDS requires some initialization parameters. This parameter refers to a list of addresses the discovery process checks for new publications and subscriptions. It is an initial peer list where there could be candidates remote publications and subscriptions. The format of this discovery peers parameter is shown in figure 9.b.

Qos Library Name Data Distribution Service provides a mechanism for setting QoS policies to entities based on XML configuration files. Those files contains libraries of QoS profiles that can be used.

Qos Profile Name This parameter specifies the QoS profile to be used that is stored on the QoS library.

The configuration window of a DDS account for this parameters is shown on figure 9.a.

4.3 DDS Bots

The DDS Bot is the interaction point of the user to the Data Distribution Service. When publications or subscriptions are available, the application shows a bot on the contact list that can access to them via conversations.

A topic can be described by its name and its type description. A DDS bot is specialized in a specific topic based on their topic name. With this approach, although we have topics with the same topic type but different topic names, a DDS bot is created for each of these topics. Then, a DDS bot represents the association between a DDS Topic and a buddy in the contact list of an Instant Messaging client.

Figure 10 depicts how the mapping between discovered topics and DDS bots is done. Each discovered topic is associated to a unique DDS Bot according to the topic name. In case of name conflicts (same name, different type), only the first one is taken into account.

Figure 10: Mapping between topic and DDS bots
A DDS bot is represented in the application as one buddy in the contact list of the instant messenger client. The buddy name is the name of the topic that controls the bot.

The conversations maintained with a DDS bot are always related to the DDS topic associated with the bot. For example, if the user asks a bot to be notified on DDS events like sample arrivals, this bot is going to notify the user on sample arrival for the DDS topic associated to this bot. DDS topics and DDS bots are associated by their name.

5 Presence and Status

[Describir el uso de políticas de calidad de servicio de DDS para el control de presencia y estado de los bots]

Most of the instant messagery services present today support presence control and status. Presence control is an emerging technology that tries to integrate the instant messaging with other services [19, 20].

The use of user statuses in instant messenger services is common for almost every instant messenger service. When an user connects to an instant messenger service it is possible to define a flag that indicates the availability (status) of this contact. The most common statuses used are online, offline, away and not available. Each of this statuses are usually associated with an emblem that simbolizes the nature of the availability. For example the “wrong way” signal is usually used for not available status, and the “clock” emblem is used in away statuses.

The number and kind of status available depends directly of the instant messenger service implementation. The use of statuses can help the presented DDS application to display visual information about remote publication and subscriptions. This section is aimed to explain how the status capabilities of instant messenger clients are used within our DDS application.

5.1 Visual status information

The evolution of the internet technologies has made that more sophisticated methods of communication had been incorporated into instant messenger services. For example, now it is possible to experience video conference within the instant messenger service (Microsoft Live Messenger, skype), call forwarding when the user is not available, birthday notification,... Normally, all those capabilities are not always available and sometimes depend on their status.

To notify the user about these capabilities or extended status information, the instant messenger clients make use of emblems, that are small icons that can be associated to buddies in the contact list of the user. The emblems used and their meaning depends on the instant messenger service used.

As mentioned before (sec. 2.1), in Data Distribution Service there are a lot of events taking place concurrently that informs about remote situations (such as new publications, data loses, ...). These notifications usually are transmitted to the user by a textual conversation with a bot. In some DDS scenarios, the amount of notifications can make the textual information innapropriate. Reading text takes some time to the user, and if
there is a lot of notifications, then a lot of text should be read by the user about this notifications. To avoid this problem, the use of visual information has been included.

In figure 11 shows the emblems that have been chosen for representing the different notifications to be made.

### 5.2 Textual status information

In some cases the presence and status of a buddy is also associated with some extra information is shown beside the display name of the buddy. In a regular instant messenger service, a user use this area to show some relevant information that considers of interest for other contacts [21]. It is usual when connecting to an instant messenger service see many users using their display name as a point to broadcast information about their mood, or recommend an interesting website or youtube video.

The idea used in this work is to use this normalized behaviour to broadcast some information relative to publications and subscriptions without the need of establishing a conversation explicitly.

The display name is used to denote information about remote entities for a certain topic. At a short glance, the user can read in the display name the number of remote entities that are participating publishing or suscribing to a certain topic in a Data Distribution Service domain.

In the contact list figure 12 an example of using the display name information for broadcasting information about remote entities.
6 Application Deployment

A Data Distribution Service based application consists of a bunch of nodes that are distributed over a network to exchange information between them. The information is passed between the nodes as topics being published on domains.

The main when deploying a DDS application is that the entities that should communicate using publication/subscription mechanism have to know the domain where the topics are published.

The DDS instant messenger application, as described in section 4.2, is organized by DDS accounts. Each of those accounts has a Domain Participant that is present in one domain.

The application configuration and the number of accounts necessary to configure and logon depends on the scenario in which we want to install the application. For example, if we want to participate in 3 different DDS domains at least you need three DDS accounts (one assigned to one domain).

An unique Publisher and Subscriber is shared between all the Data Readers and Data Writers associated to a certain domain, so if the user want to participate in the same Domain with different QoS parameters for Publisher and Subscriber, a different DDS account should be created for each of this different profiles. Other use case scenario for multiple DDS accounts for the same domain is to use different discovery peer lists in one of each accounts.
Algorithm 1 StockNews IDL definition

```plaintext
struct StockNews {
  string <4> stockTicker;
  string <128> stockNewsUrl;
  string <128> updatedBy;
  string <128> updatedDate;
};
```

7 Examples and Results

7.1 Examples

To illustrate the presented project in action, this section is going to describe some use cases. For testing purposes, the StockNews example from the RTI Data Distribution Service middleware implementation is going to be used in this section.

The StockNews data definition is described in listing 1. For the whole testing process, remote publishers for StockNews topic are going to be run to see how they interact with our application.

7.1.1 Connection to a dds domain with an account

First of all, some parameters have to be taken into account before running the application. The domain Id and profiles should be configured before starting the application. They can be configured in the Account Editing dialog on the application.

![Figure 14: DDS Account configuration process](image)

The remote publisher runs on domain 0, and default QoS settings, so the defaults settings in the instant messenger client are enough for our purposes.
7.1 Examples

7.1.2 Set a QoS via a profile

In some cases the default QoS settings are not valid because they can result in “incompatible qos” when trying to match remote entities. That’s the reason because the QoS settings can be changed dynamically in our program, to fit this applications.

In this example the default profile for the participant is set to the value “ExampleProfile”, which is contained in a XML file in the current directory of the application. After this moment, all the entities created below the current participant, are going to be created using this QoS Profile settings.

![Figure 15: Setting filtering configuration](image)

7.1.3 Set notification filtering

The user can specify the events he wants to be notified. These events includes discovered entities, sample arrival, QoS infringements... In the figure 15 the user asks to be notified about all the events that could happen in Data Distribution Service.

In the screenshot it can be noticed that the sample arrival notification only is done after the user request explicitly to be notified about this event.
7.1 Examples

7.1.4 Asking about entity status

When many events occur in *Data Distribution Service* user might be confused about the status of the system. To solve this problem, the user can ask explicitly about the status of a local entity.

In figure 17 the user asks about the current DataReader status, and the system suma-
7.2 Results

The present project is a proof of concept of how instant messenger services and Data Distribution Service can be joined together in an application to take the advantages of both:

- the ease of use of the instant messenger paradigm
- the power of easy data dissemination because the Data Distribution Service

The main problem of Data Distribution Service is the complexity of the system for the new user. The great amount of different concepts like complex Qos policies that DDS involves, makes the learning process hard at the very first phases. In the other hand, we have the instant messenger paradigm, that is so familiar for the user that makes the system simple.

The joined efforts of both technologies allow the user to access Data Distribution Service but with a lower learning curve. The user can interact with Data Distribution Service with no more difficult than asking some friend about yesterday’s match.

Not all the possible applications using Data Distribution Service are suitable to be used with the presented instant messaging application. It is because the types of data that can be used as a topic. Very complex types of data that can be used as a topic in DDS cannot be directly translated in human readable data. Although these topics are supported by the application, the sample data exchange with the user might be very confusing. This is not a program design fault, it is directly derived from the nature of the data used in some use cases.

In most cases the instant messenger approach to DDS presented in this document can be used as:

**Testing Tool** The application is valid to be used as quick testing tool, as is it possible to check the publications available using an specific QoS profile.

**Monitoring Tool** The instant messenger approach can be used as a monitoring tool, as it makes possible to obtain information about the present remote entities and their status. It is also possible to identify some communication problems, as it is possible to look into low level protocol statistics.

These use cases have shown the utility of our systems where a quick general interaction with Data Distribution Service is needed. Obviously, the work presented here doesn’t fit in all possible scenarios constructed over Data Distribution Service middleware. Some scenarios can’t be handled by this approach, for example where thousands of samples are being published by second. In this scenarios should be treated as an special case, as they couldn’t be handled by the using of visual or textual interfaces by humans as the humans aren’t able to process the information.
References


[16] Badgerfish: Translating xml to json.

Javier Povedano Molina 42 Universidad de Granada


