

A new design for the Tango event system

May 20, 2010

1 Introduction

Today (up to Tango release 7), Tango event system is based on the CORBA notification service. Tango uses omniNotify as CORBA notification service implementation. One event is detected by the device server process and pushed to the notification service. It's the rule of this notification service to forward the event to the interested clients. After several years of usage, it appears that this system has some limitations:

- The notification service implementation uses by Tango (omniNotify) is a "dead" software. There is no new release nor bug fixes since several years and the mailing list is remarkably silent.
- The communication between the device server process, notification service and clients are classical CORBA (GIOP/IIOP) calls and therefore point to point communication. In case of several clients interested by the same event, the notification service will send several times the same event on the network (one time for each interested clients)
- The event data is sent to the notification service and to the clients using CORBA Any objects. This adds unavoidable memory copy in both the server side and the client side. The use of CORBA Any object also increase the amount of data which are transferred on the network because information identifying the type of the data within the Any has to be transferred. This is a limited factor in term of performances
- The notification service needs a large amount of memory because it is a highly threaded process (stack size for each thread) and because in some cases, it has to do event buffering.

In order to remove these limitations, we are actually looking for another solution to implement the Tango event system. Three solutions have been investigated:

1. A 100 % Tango based solution using Tango group
2. Using the OMG Data Distribution Service (DDS)
3. Using the Zero MQ protocol

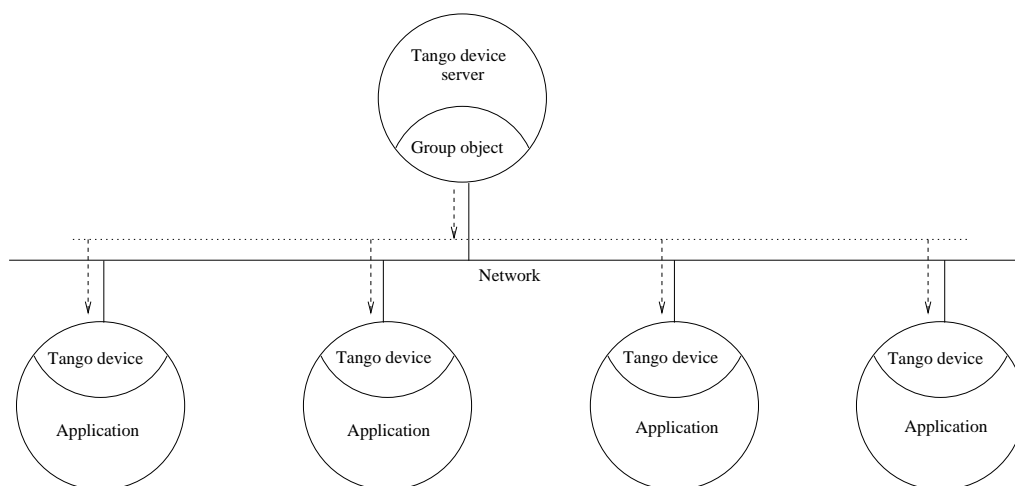
This paper reports on these solutions with their respective advantages and drawbacks.

2 Tango event using Tango group

2.1 Basic ideas

Sending one event from a device server process to a client means that the client acts as a server and the device server process as a client. It is the device server which initiates the communication and the client which receives the data. In this solution, the idea is to transparently have a Tango device in the client application. When an event is detected in the device server process, the event

propagation will be done by one remote invocation of one of the method supported by the Tango device embedded in the client application. Event are very often propagated to several clients. Within such a solution, this means executing the same remote invocation on a group of Tango devices. Within Tango API's, we already have Group object which allow Tango communication (command invocation or attribute reading/writing) between a client (the device server process in this case) and a group of Tango devices (the client applications). This is summarized in the following drawing.



Tango communication are unicast. In case of several applications interested in the same event (see previous drawing), this event will be sent on the network several times (as many times as they are applications interested in the event). Nevertheless, the Tango Group object work in two phases: First, it sends the request to all its members using asynchronous call and then it fetches all the reply coming from its members. This allow some kind of parallelism between group members.

2.2 Implementation

Let's assume that the Tango class for the Tango device embedded in each application listening for event is called `_EventReader`.

2.2.1 Client application subscribing to an event

The following actions have to be done when an application subscribes for the first time to an event:

- Create in database one entry for the `_EventReader` Tango class with a transient Tango device.
- Start this Tango class (this will export the transient Tango device to the database)
- Inform device server that the application subscribes to the event. This is done using the existing `EventSubscriptionChange` command of the Tango admin device running in the device server process. Send the transient Tango device name to the server

The device server admin device will have to

- Create the Tango Group if it does not already exist.
- Add the transient device to the Tango group

2.2.2 The `_EventReader` Tango class

This class is not a classical Tango class. Even if event propagation could be done using command and dynamic attributes, we could have simpler and faster implementation if we create a dedicated network interface for devices of this class. Nevertheless, this new network interface will inherit from the IDL interface `Device_2` and therefore, device belonging to this class will also be Tango device. The network operation defined in this new network interface will be:

1. void `write_event`(in String event_name, in DevVarDoubleStringArray double_filters, in AttributeValue_4 event_data); It will be used to receive the change, archive, periodic and user events
2. void `write_att_conf_event`(in String event_name, in AttributeConfigList_3 event_data); It will be used for the attribute configuration change event
3. void `write_data_ready_event`(in String event_name, in AttDataReadyList event_data); Used for the data ready event
4. void `write_heartbeat_event`(in String ds_name); Used for the device server process heartbeat event
5. void `write_generic_event`(in String event_name,in DevVarDoubleStringArray double_filters, in Any event_data); For future extension. This call will allow us to add new event type if required without redefining this IDL interface.

These calls will allow an implementation which is very close to what we have nowadays in the `push_structured_event()` network operation defined by the CORBA notification service.

On top of these 5 network operations, this class will have few classical Tango commands and/or attributes only dedicated to event system debug purpose. I do not see any reason for other Tango class to use this network interface. It will be used only by the `_EventReader` class. Change in this interface will be synchronized with change in the `Device` IDL interface in order to simplify release compatibility problem.

2.2.3 The `EventGroup` class

Obviously, the `Group` class does not know these new network operations. A new class `EventGroup` will be written. This class will inherit from `Group`. It will have methods to invoke the 5 new network calls in a "group way".

2.2.4 The `EventDeviceProxy` class

In the same way, we have to write an `EventGroup` class, we will have to write a `EventDeviceProxy` class. This class will inherit from the `DeviceProxy` class and therefore, will benefit from all the reconnection algorithm implemented by the `DeviceProxy` class. Methods allowing an asynchronous execution of the 5 methods described above will be provided by this new class. The `EventGroup` class will have `EventDeviceProxy` instance as member of its group (instead of classical `DeviceProxy`).

2.2.5 Firing an event

Within such a framework, firing one event means calling the right method on the `EventGroup` object. With a proper management of the error returned by the call, it will be possible to remove from the `EventGroup` instance the transient device associated to a crashed application.

2.2.6 Application shutdown

If an application has used events, in the function which is executed during the application exit, the application transient device will be removed from the database. In case of application crash, the transient Tango device will stay into the database. A garbage collector thread could be implemented in the database server in order to delete from the database "device" table entries for transient devices which do not answer to ping request.

2.2.7 Naming

The Tango class implementing the event receiver is named `_EventReader`. In order to register the transient device in the database, we also need to define a "pseudo" device server name (`exec_name/instance`) and a device name. These names could be:

- `_EventReader/host_pid` for the "pseudo" device server name
- `_event/host/pid` for the transient device name

With Java, it is not possible to get the virtual machine pid. Another naming convention will be used but close to this one.

2.3 Performance

Obviously, the code is not written. Nevertheless, to evaluate what could be the performances achieved with such an implementation, a Tango device class with an attribute of `DevEncoded` data type has been used. A small client using the actual `Group::write_attribute()` method has been used to write this attribute in 1 to 10 different device server process embedding the test Tango class.

2.3.1 Hardware used

The computers used are described in the following table:

Host name	OS	Memory	Processor	Frequency
eclipse	Ubuntu 9.04 (64 bits)	4 GBytes	Intel Core2 Duo	2.66 Ghz
pcantares	Ubuntu 9.04	1.5 GBytes	Intel P4	2.4 Ghz

All these computers are connected on the same sub-net. The network connection speed is 100 MBits/sec (12.5 MBytes/sec)

2.3.2 Test set-up

The measurement has been done for 3 different size of events:

- 1 Tango `DevLong` (32 bits data) is transferred within the event
- 1024 Tango `DevLong` (4 KBytes) is transferred within the event
- 262144 Tango `DevLong` (1 MByte) is transferred within the event

In the first case (1 `DevLong`), the amount of data added by the Tango layer (attribute name, quality factor...) is not negligible. The tests have been done with an amount of data comparable to a real Tango attribute event (92 bytes instead of 4)

2.3.3 Using remote hosts

The client is running on the host called *pcantares* and the device server processes run on *eclipse*. Based on the measurement done for the May 2009 Tango meeting at ESRF, the results are the following:

Clients	1 DevLong		1024 DevLong		262144 DevLong	
	Tango V7	Group	Tango V7	Group	Tango V7	Group
1	770	1330	650	800	10.7	8.5
2	770	1320	460	570	5	4.3
5	400	870	200	280	2.3	1.7
10	220	450	100	150	0.9	0.8

From this table, it is clear that we have an increase in term of performances except when the amount of data to be transferred is large.

2.3.4 Device server and clients on the same host

The computer used for this test is *pcantares*. Tango 7.1.1 has been used. The results are:

Clients	1 DevLong		1024 DevLong		262144 DevLong	
	Tango V7	Group	Tango V7	Group	Tango V7	Group
1	620	2500	600	1830	40	33
2	420	1250	400	930	30	16
5	230	520	210	380	15	7
10	170	250	160	190	8.5	3.3

3 Multicasting for Tango event propagation

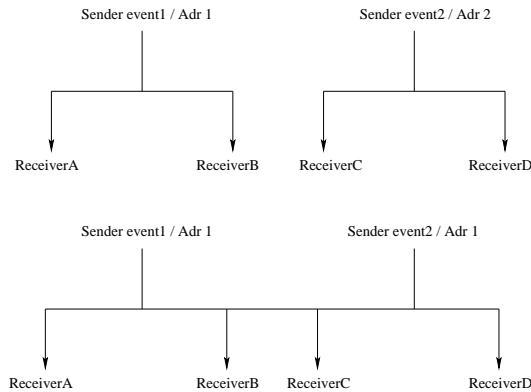
3.1 Multicast

Multicasting is a way for a computer to send data over a network to a group of receivers. A receiver interested in the data sent using multicast has to first join the multicast group. For multicast, the IP protocol uses class D network address from 224.0.0.1 to 239.255.255.255. However, the address range between 224.0.0.1 and 224.0.0.255 is reserved. There are still 265 236 975 addresses available. Host uses the IGMP protocol when they join or leave a multicast group to inform routers of their interest in the group. With the classical TCP/IP stack, multicasting is available on top of IP or UDP. UDP is a non-reliable protocol and multicasting simply done over UDP is also non-reliable. If you need reliable multicast communication over UDP, you need a protocol which will manage packet re-ordering and re-transmission in case of losses.

3.2 Multicast in Tango event system

3.2.1 Multicast groups

Multicasting seems to be an ideal solution for Tango event propagation where an event is detected by a Tango device server process and then propagated to all the interested clients (on the same host or on remote host(s)). If all the interested clients belongs to the same multicast group, they will receive the event by a single network transfer. Nevertheless, there is a drawback when using multicast with several events.



On top of previous figure, you have the ideal case where the sender of event1 uses one multicast group with Adr1 and the sender of event2 uses another multicast group with Adr2. The receivers of event1 (ReceiverA and ReceiverB) receive only event1 while the receivers of event2 (ReceiverC and ReceiverD) receive only event2. At the bottom of the same figure, both senders use the same multicast group with Adr1. Now ReceiverC and ReceiverD receive event1 while ReceiverA and ReceiverB receive event2. The unwanted events have to be filtered out by the software but this generates a useless load on the NIC (to receive the packets for the unnecessary events) and on the CPU (to filter out these data).

In an ideal Tango system, we would need as many multicast group address as events used in the system. As soon as we use the same multicast group for several events, we will have the "host pollution" describes above. It is possible to use several multicast group addresses within a Tango event system but we have to find a solution to "equally" spread the events on the available multicast group addresses in order to minimize this "host pollution".

3.2.2 Port number

With distributed software like CORBA, by default, server processes choose a random port number. When using multicasting, all the server member of a multicast group has to listen on the same port if they want to receive the data. Therefore, it is not possible to use random port number because there is no guarantee that the same port number is free on all the concerned computers and even worse if case of late joiner host. If you use multicasting, you have to use fixed port number.

4 Data Distribution Service (DDS)

The Data Distribution Service is an OMG specification. Today it is in its release 1.2. In the same way that OMG has defined GIOP/IIOP for CORBA interoperability, OMG has defined the DDSI protocol (a reliable UDP multicast protocol) for DDS implementation interoperability. When you use DDS, you are not interested anymore in remote object interface (methods). You only defined the data which will be exchanged in your system. This definition is done using the CORBA IDL language. Then, you write data publisher (writer) and subscriber (reader). Today, there are at least 3 implementations of DDS:

1. OpenDDS (<http://www.opendds.org/>)
2. RTI (<http://www.rti.com/>)
3. OpenSplice from PrismTech (<http://www.opensplice.org>)

OpenDDS is using TAO during its connection establishment process. Therefore, it is not really suitable for Tango. RTI is a commercial implementation and therefore also not suitable for Tango. OpenSplice is since beginning 2009, an open source implementation. It is written by the PrismTech

company. It comes with their own protocol (UDP reliable multicast) for network communication even if they also support DDSI. It is the implementation which has been tested. All the tests have been done using their protocol and not DDSI.

4.1 OpenSplice studies

4.1.1 Getting OpenSplice (binary and source distribution)

OpenSplice is available from a PrismTech Web site at the address <http://www.opensplice.org>. It is down-loadable in binary or source format. In binary format, it is available for Linux x86 32 and 64 bits, Windows x86 and Solaris 10 Sparc. The source download comes with its own configure (it's not the GNU auto-xxx tools). Some compilation instructions are available. Following these instructions, I was able to compile and install a debug release of DDS (on Ubuntu 9.04 box) even if I still don't know how I can specify the install directory. The source distribution contains DDS for C, C++ and Java.

Tests have been done using **OpenSplice 4.3**. Due to our support contract and to some complains sent to the support group, I have received a release 4.3.1 incorporating several bug fixes discovered by us and other. Since two weeks a new release 5.1 is available for download.

4.1.2 Working out of the box?

It was not possible to get reliable communication between a OpenSplice publisher and subscriber during our first tests. We finally bought 2 days of expertise from one OpenSplice expert from PrismTech. It appears that the ESRF network is not very good (confirmed during the next product tests) and that the reliable protocol has to send some repair data packets. We had to tune some buffers size in the OpenSplice configuration to correctly send the classical and repair data sockets.

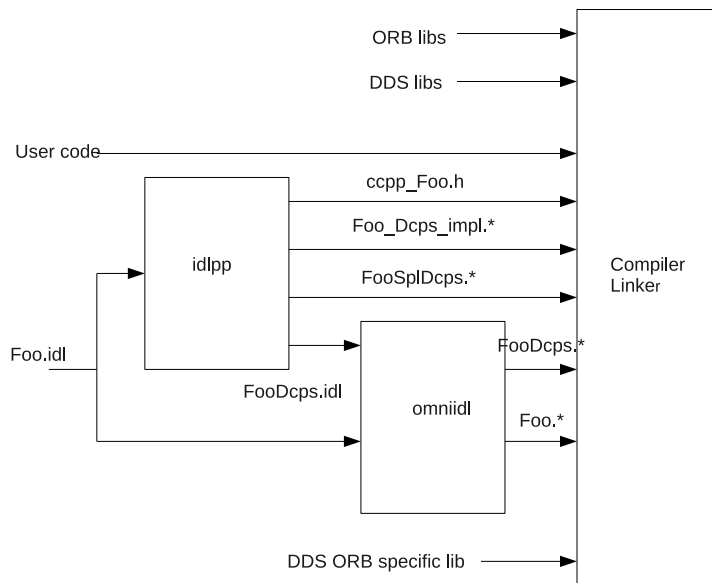
4.1.3 CORBA ORB / OpenSplice cohabitation

OpenSplice support 2 modes:

- The *standalone* mode where the application is only using DDS
- The *ORB integrated mode* where the application is used with an ORB as well as with DDS. This is the Tango case.

In a standalone context, OpenSplice DDS provides, apart from the DDS related artifacts, all the artifacts implied by the IDL language specific mapping. In an ORB integrated context, the ORB pre-processor will provide the artifacts implied by the IDL language specific mapping, while OpenSplice only provides the DDS related artifacts. The application data type representation provided by the ORB is also used within the OpenSplice DDS context. In this way, application data types can be shared between the ORB and OpenSplice within one application program. For instance, a sequence created by OmniORB can be transferred by an OpenSplice topic. Without this ORB integrated mode, there is absolutely no guarantee that the IDL sequence class is internally coded the same way between OpenSplice and OmniORB.

OpenSplice and omniORB idl compiler usage The following figure shows how you have to use the OpenSplice and omniORB IDL compilers when you want to use the OpenSplice integrated mode.



For a DDS Topic/CORBA interface defined in a file `Foo.idl`, the files generated by the IDL compilers are:

- `ccpp_Foo.h` which is the master include file for DDS
- `FooDcps.idl`: The IDL definition of the topic type specific classes (`FooTypeSupport`, `FooDataReader`, `FooDataWriter`)
- `FooDcps.h` and `FooDcps.cpp`: The topic type specific classes
- `FooDcps_impl.cpp` and `FooDcps_impl.h`: The implementation of the topic type specific classes
- `Foo.h` and `Foo.cpp`: The Topic and ORB interface classes files
- `FooSplDcps.h` and `FooSplDcps.cpp`: Files for DDS (un)marshaling

Once, these files are generated, they have to be compiled and linked with:

1. The user code
2. The DDS libraries
3. The ORB libraries
4. A DDS ORB specific library. This library has to be generated for your ORB of choice. The OpenSplice distribution comes with few DDS-ORB specific libraries already compiled but not for omniORB. Following a recipe given by OpenSplice, this library has been successfully generated for omniORB 4.1.4 for Ubuntu 9.04 in 32 and 64 bits

A small client/server example has been written using this ORB integrated mode. The client is first publishing some data using DDS and then call an IDL operation on a remote CORBA object. The data sent as the IDL operation input parameter are the data used during the topic publication. The server implements a DDS subscriber and a CORBA server as well. This works fine.

4.1.4 OS signals, garbage collection and process crash

In its release 4.3, on a Linux box, OpenSplice install signal handler for 4 signals which are:

1. SIGINT
2. SIGHUP
3. SIGQUIT
4. SIGTERM

Tango also install its own signal handler for these signals. Special care has to be taken in the signal management if DDS is used within a Tango server. In its release 4.3.1, OpenSplice also install signal handler for signals which by default abort the process generating a core file (SIGILL, SIGABRT, SIGFPE,.....) The remaining signals are left to their default behavior. When a process using OpenSplice 4.3.1 receives such a signal, the OpenSplice signal handler shutdown all the DDS entities but **does not generate a core file** and do not print any message like "seg fault". This could be annoying in case of debugging a device server process after a process crash.

4.1.5 Heartbeat

In order to be informed in case of process crash, a Tango device server sends every 9 seconds a heartbeat event. A dedicated thread in the event subscriber checks that these heartbeats events are still received. In case they are not, an exception is sent to the user code to inform it that the device server process has crashed. This is a mandatory feature for an event-based system.

Within its QoS set, DDS has the Liveliness QoS which in its so-called Automatic mode is doing exactly this. The subscriber is informed when the publisher becomes alive or when it disappears because it has been shutdown, killed or crashed. This is a nice feature and it could remove quite some lines of code in the Tango core. This feature works well with OpenSplice 4.3.1 but not with the release 4.3! In release 4.3.1, this is achieved by signal handlers which shutdown all the DDS entities when they are executed. OpenSplice 4.3.1 install their own signal handler for all the signals which by default aborts a process with or without generating a core file. This is the trick. Nevertheless, **there is no protection against killing a device server with a "kill -9"** command. In this case, the subscriber is not informed that the publisher has gone. The following sentences comes from the OpenSplice Deployment guide:

The cleanup mechanisms are not executed when an application is terminated with a KILL signal. For this reason a user must not terminate an application with a kill -9 command (or, on Windows, must not use TaskManager's End Process option) because the process will be forcibly removed and the cleanup mechanisms will be prevented from executing. If an application is killed in this manner, the shared memory regions of OpenSplice will become inconsistent and no recovery will then be possible other than re-starting OpenSplice and all applications on the node.

I personally find this rather annoying !

4.1.6 Content Filtered Topic

A content filtered topic is the way to create a Data Reader attached to a topic on which you filter the topic data using a sub-set of the SQL language.

At a glance, we can say that within Tango, we will have one topic for each event type. Within the topic data type, we will have the full event name as a string (something like tango://kidiboo:10000/my/device/nam). On top of being used as the Topic key, this name field can be used in the SQL expression to create the content filtered topic. This works fine except that the Data Reader receives an invalid sample when any publishers shutdown (even for those which published topic normally filtered out!!). This is identified by PrismTech as a bug and here is the support answer related to this behavior: *This is a bug that is already in the process of being fixed. It is expected to make it into the next release.* We will see.

It has to be noted that the filtering is done by the host receiving the data. This means that even with content filtered topic, all the hosts with subscribers listening on the topic will receive the data but only the application with the right filtering will be informed. In a Tango world with one Topic for the change event, it means that all the hosts where one application listening for change event (on any attribute) will receive all the change events (for all attributes on which events are used). Then the OpenSplice layer will filter only the change event the application is interested in but... It's clear that if DDS is used within Tango, we will have to deal with the so-called partitions to deal with this problem. I think this kind of behavior is not due to DDS itself but rather to the multicast usage.

4.1.7 Removing / Modifying a Topic

A Topic has three elements:

1. Its name
2. Its data type (defined in CORBA IDL)
3. Its QoS

Once a topic is registered in the system, it is very difficult to remove it. Here is the answer I received when I have asked about this point to the mailing list:

In accordance with the specification, there is indeed no API that ' globally*' destroys topics (and their related types) as one of the key-features of DDS is to maintain and provide information even for late-joining applications. The issue with a programmatic interface to 'wipe' this 'topic-memory' in DDS is that its almost impossible to specify its behavior cq. implications if there are still applications running or even "to be started as late-joiners" that rely on the topic-definition. Please note that this 'issue' is NOT related to any specific implementation of DDS (and/or usage of services/daemons providing full coverage of all DDS-profiles) as it doesn't make a difference if the '*knowledge*' about topic/types is 'embedded' in running-applications, persistent-storage, middleware-services and/or even late-joining applications (that still utilize an 'old' data-model). So right now the way to start with a clean-sheet is to stop (and later update) any applications that 'use' the old data-model as well as restart the infrastructure that is supposed to '* remember*' this information for late-joining applications (so also requires deletion of persistent data-stores before restarting the infrastructure).*

We are considering a product-extension where topic description (and related types) would be sort of 'recounted' and could be purged (automatically or by an API) when there's no usage anywhere in the system anymore (i.e. all related data has been disposed, including from the transient/persistent stores). Note that even then, there is a risk that when re-activating a system that includes previously unactive nodes that still have persistent 'knowledge' of the 'old' data, it could be avertedly 'revived'.

Therefore, I think it is a good idea to add in the Topic name a release number which could help us to have at the same time several Topic definitions (if required) for our events. Topic name could be something like "TangoChangeEvent_V1"

4.1.8 Use of the builtin topics

OpenSplice manage the 4 built-in topics defined in the DDS spec. They have added a fifth one named "DCPSheartbeat" which could be interesting for us in a tool like astor to rapidly check that OpenSplice run nicely on our hosts.

4.1.9 PrismTech tools

PrismTech has 2 tools:

1. The ospl configuration editor tool which helps to configure OpenSplice

2. The ospl tuner tool which helps to tune / understand your system

The ospl tuner works fine and is really a good tool even if in our edition (Compact), it is only able to deal with the DDS entities which runs on the local host. If you want to see DDS entities on another host, you need another OpenSplice service which is not part of the Compact edition.

The ospl configuration editor is also a good tool even if its usage is not mandatory (could be replaced by a simple text editor).

4.1.10 OpenSplice and the "dummy0" network interface (ESRF specific !)

In case of computer with several NIC, to select on which interface the multicast enabled socket has to be connected, OpenSplice uses the "setsockopt()" system call with the IP_MULTICAST_IF flag giving to this call the interface unicast address. This interface unicast address is defined in the OpenSplice configuration file. In our RedHat computers (Release 4 or 5), we have 2 network interfaces with the same IP unicast address: the classical *eth0* interface but also a *dummy0* interface (I have no idea why we have this dummy0 interface). The setsockopt system call associates the socket to the dummy0 interface!! If you do not call this setsockopt() function, the socket will be associated to the network interface on which the default route has been defined (*eth0* in our case). Commenting out the setsockopt system call in the OpenSplice software temporary solves this problem. Nevertheless, this is not a solution in case of OpenSplice use in Tango !!

After several tries on another computer (queg) on which I could play with the network interfaces, it appears that the interface returned by the "setsockopt()" system call (with IP_MULTICAST_IF) depends on which interface has been started first. If the *dummy0* interface has been started first, then it will be the returned interface. To run OpenSplice on a computer configured as described previously, we need to find out in which file is described the order in which interfaces are started. The interface *eth0* has to be started first.

4.1.11 Host on several sub-net

Works fine once you have set-up correctly the necessary parameters in the configuration file. Nevertheless, it requires that the router between the different sub-net is doing multicast routing. The router configuration has been changed by the network group. Only the multicast groups used during our test (226.10.11.12, 226.20.21.22 and 226.30.31.32) have been allowed to be routed between sub-net.

4.1.12 A slow host (10 Mbit network) in the multicast group

TODO

4.1.13 Re-connection

Simple Client / Server re-start Several cases have been studied:

1. Publisher stopped and re-started with a subscriber waiting for topic instance: OK
2. Publisher crashed and re-started with a subscriber waiting for topic instance: OK
3. Publisher killed with -9: OK

DDS started after the clients/servers If a publisher (server) or a subscriber (client) is started before OpenSplice, the call to create the DDS Domain Participant will fail. It is possible to detect this error and to -implement some re-connection mechanism until the OpenSplice software is started.

No error reported by the publisher write() method if OpenSplice is shutdown !!!

Using a Topic with liveliness QoS set to automatic and a 5 seconds dead time, the subscriber is warned only after 1 minute!!! When OpenSplice is re-started on the host where the publisher is running, the publisher software do not re-connect. It is needed to stop/start the publisher process. The subscriber process automatically re-connects.

Network cut Seems to work but after a long delay: 1 minute before the host where the subscriber is running is warned by an invalid data (NOT_ALIVE_DISPOSED) (and a liveliness event only during the first de-connection) plus another dead time before the effective re-connection. **No error** reported by the Data Writer write() method while the subscriber host is disconnected. This dead time of 1 minute is tunable with the parameter Lease/ExpiryTime in the xml configuration file. The default is 60 seconds. The more you decrease this value, the more heartbeat (and associated processing time) you will have to handle this.

If you modify this Lease/ExpiryTime to 20 seconds for instance, I had a faster notice that the host is disconnected but nothing happened when I reconnect it !!!

Host reboot / crash The publisher is running on Host A and connected to a subscriber on Host B. Host B is re-booted. **No error** reported to the publisher when it writes Topic instances while the subscriber is down. When OpenSplice and the subscriber process re-start on Host b, the publisher re-connects fine.

Host crash simulated using a laptop without battery and suddenly un-powered. Idem previous case.

4.1.14 Release compatibility

Here is the answer I got from the support team about compatibility between OpenSplice releases:

We do not support backwards compatibility between OpenSplice versions. You should always perform a complete IDL and code recompilation when switching OpenSplice version. Also we strongly recommend using the same version of OpenSplice across all your system nodes. Using mixed versions across a single system can lead to un-predictable behaviour as the OpenSplice code will not be the same on all the nodes.

This could make our life difficult in case of large control system. This means that in case of OpenSplice new release, we have to re-compile Tango libraries, all the device server processes and we need to install in all hosts used in the system the new OpenSplice software. This is may be something doable for a small control system but it is a problem for medium size and large control system.

4.1.15 What about Java?

A Simple example works fine. Nevertheless, it has to be noted that this **is not pure Java** implementation. It's a JNI layer above the C API.

4.1.16 What about using DDSI?

TODO

4.1.17 API

There are several calls within the API which acts as objects factory. Therefore, they return a pointer to the newly created object. In case of error, the returned pointer is NULL and **there is no other error code**. This means that you have no information about why the call failed. This is annoying!

The DDS API is defined in the OMG document in the IDL language. Some types (called xxx_TYPE_NATIVE) are intended to be defined in a native way. Even if it is written in the specifications that these types are opaque types and therefore the actual definition of the type

does not affect portability, it is not true. For instance the DomainId type used in the first call an application is doing is a string in OpenSplice and one integer in RTI!! **This generates compatibility problems.**

4.2 Performances

4.2.1 Hardware used

The computers used for the benchmarking of DDS are described in the following table:

Host name	OS	Memory	Processor	Frequency
pbruno	Vista SP2 (32 bits)	4 GBytes	Intel Core2 Duo	2.66 Ghz
pctaurel2	Vista SP2 (32 bits)	4 GBytes	Intel Core2 Duo	2.66 Ghz
lapseg2	Ubuntu 8.04	1 Gbytes	Intel pentium M	1.7 Ghz
eclipse	Ubuntu 9.04 (64 bits)	4 GBytes	Intel Core2 Duo	2.66 Ghz
l-cb032-1	Ubuntu 9.04	3 GBytes	Intel Pentium D	3.2 Ghz
pcantares	Ubuntu 9.04	1.5 GBytes	Intel P4	2.4 Ghz

All these computers are connected on the same sub-net. The network connection speed is 100 MBits/sec (12.5 MBytes/sec)

4.2.2 Test set-up

A topic similar to what is today a Tango event (change or archive event) has been defined. Two small programs have been written. One acts as a DDS publisher (similar to a Tango device server sending the event) and the other one acts as a DDS subscriber (similar to a Tango clients receiving the events). The measurement has been done for 3 different size of events:

- 1 Tango DevLong (32 bits data) is transferred within the event
- 1024 Tango DevLong (4 KBytes) is transferred within the event
- 262144 Tango DevLong (1 MByte) is transferred within the event

The publisher publishes the data as fast as it can and the subscriber reads the data. A check on lost event is also implemented. The OpenSplice software and the DDS QoS have been tuned in order to have a reliable and sustainable communication. The OpenSplice shared memory segment size is 33 MBytes.

4.2.3 The results

DDS compared to alternative solutions The publisher is running on the host called *pcantares* and the subscriber runs on *eclipse*. Based on the measurement done for the May 2009 Tango meeting at ESRF, the results are the following:

Clients	1 DevLong			1024 DevLong			262144 DevLong		
	Tango V7	Group	DDS	Tango V7	Group	DDS	Tango V7	Group	DDS
1	770	1330	12500	650	800	1850	10.7	8.5	7.9
2	770	1320	10500	460	570	1800	5	4.3	7.9
5	400	870	7900	200	280	1800	2.3	1.7	8
10	220	450	6500	100	150	1700	0.9	0.8	8.1

From this table, it is clear that we have a big increase in term of performances with DDS. We also clearly see the multicast effect with a constant performances whatever the clients number is (except for transferring a small amount of data).

DDS on several hosts The same kind of test has also been done with the same host used as publisher but subscriber(s) running on different hosts. The results are:

Clients	1 DevLong			1024 DevLong			262144 DevLong		
	eclipse	lapseg2	pctaurel2	eclipse	lapseg2	pctaurel2	eclipse	lapseg2	pctaurel2
1	12500	8500	12000	1850	1850	1830	7.9	7.8	7.9
2	10500	5700	10600	1800	1900	1850	7.9	7.8	7.9
5	7900	2950	8000	1800	1900	1800	8	7.7	7.9
10	6500	1500	5800	1700	1250	1800	8.1	7	7.7

The host *pctaurel2* is running Vista OS. No special problem has been detected to run OpenSplice and the test software on this platform.

Multicasting The tests has also been done with clients running on several hosts:

- 1 client: on *eclipse*
- 2 clients: on *eclipse* and *lapseg2*
- 5 clients: on *eclipse*, *lapseg2*, *l-cb032-1*, *pctaurel2* and *pcbruno*
- 10 clients: 2 clients on *eclipse*, *lapseg2*, *l-cb032-1*, *pctaurel2* and *pcbruno*

The results are:

Clients	1 DevLong	1024 DevLong	262144 DevLong
1	12500	1850	7.9
2	8000 → 11000	1700 → 1900	7.8
5	6000 → 11000	1900	7.9
10	6000 → 10000	1750	7.9

The values concerning a 4 KBytes or 1 Mbyte of data are quite similar to the previous table where all the clients are running on the same hosts. For small amount of data transferred within the event we have on average better performance than in the previous case. This is due to the architecture of OpenSplice. The network transfer has the same speed in both cases. But in case of several hosts, we have less users processes (max of 2) which extract the data from the OpenSplice shared memory. In the previous case, we had up to 10 processes reading the shared memory. I think the difference in term of performance is simply due to CPU sharing between the different processes on each host.

4.2.4 Publisher and subscriber on the same host

In order to be able to compare the results with the test done for the May 2009 Tango meeting, the computer used for this test is *pcantares*. Tango 7.1.1 has been used for the Group case. The results are:

Clients	1 DevLong			1024 DevLong			262144 DevLong		
	T. V7	Group	DDS	T. V7	Group	DDS	T. V7	Group	DDS
1	620	2500	9500 / 5300	600	1830	9000 / 4500	40	33	150 / 91
2	420	1250	5000 / 3200 *	400	930	4500 / 3200	30	16	80 / 35 *
5	230	520	2000 / 1500	210	380	1800 / 1350	15	7	35 / 32 *
10	170	250	870 / 770	160	190	800 / 700	8.5	3.3	19 / 17 *

Here also, there is a big increase of performance using DDS.

There are two numbers for the DDS column. Why? This is because OpenSplice does not behave the same according to the presence of another host running OpenSplice. If there is only one host running OpenSplice, nothing is sent on the network and we have the numbers on the left. If there is another host running OpenSplice even if there is no pub or sub running on this host,

OpenSplice send the data on the network. I have no idea why! These are the numbers on the right side which are the numbers to be taken into consideration. Please, note that the * character followed a number means that there is a big variance in the measurements done for this case and the value given is the most frequent one.

4.3 Tango integration

4.3.1 Which Topics?

One Topic for each event type:

- Change Event
- Archive Event
- Periodic Event
- User Event
- Attribute Configuration Event
- Data Ready event

4.3.2 Partitioning

The way to solve the problem linked to multicast use and described in chapter 3 is to use DDS partitions. You publish (subscribe) your data in a DDS partition. With OpenSplice, in the configuration file, you map the DDS partition(s) to the so-called Network partition(s). Each network partition has its own multicast address. We will definitively need partitioning. If you don't want to specialize your hosts, you will have to define all the network partitions in the XML configuration files. Actually, OpenSplice does not support dynamic connection to a network partition. This means that all the hosts will be members of all the multicast groups defined for the different partitions meaning that all the hosts will see all the events!!! **This is a major drawback of OpenSplice**

4.3.3 Practical implementation

If you are using OpenSplice on a host, you have to run its daemons on this host. The minimum set-up is achieved using:

- Three OpenSplice processes
- A shared memory segment
- The XML configuration file

The three processes are a general OpenSplice daemon, their networking service and a durability service which seems to be mandatory. The shared memory segment is used for the communication between these three processes and the user process(es).

4.3.4 Shared memory size

With Tango, it is very difficult to know in advance which data will be sent/received using DDS. This is a result of the running application(s) subscribing (or not) to events. Therefore, it is difficult to estimate the size of the shared memory segment and this is still an open question.

4.3.5 Share the configuration file

Should be possible to define the file content using control system properties. Then the Starter on each host could get these properties and create the configuration file before starting OpenSplice. It's not a perfect solution:

- Every host with events needs a starter device server
- Could need a large property number
- It's a dirty trick

5 ZeroMQ

5.1 Introduction to ZMQ

ZMQ (<http://www.zeromq.org/start>) stands for "0 Message Queue". This name comes from the origin of this software which was used to develop one implementation of AMQP. It is now completely decoupled from AMQP. It is a layer that you use on top of the networking stack. It has a Berkeley socket style API (You create ZMQ socket, you bind it, you write to this socket or you read from it). For ZMQ, the transported data is a message and they do not provide any encoding / de-encoding API. The messages are sent and received asynchronously. It supports different messaging patterns like point to point, publish - subscribe, request - reply... ZMQ supports multicasting when using the publish - subscribe messaging pattern. It includes the Pragmatic General Multicast (PGM) protocol defined in RFC 3208 and implemented by the OpenPGM (<http://code.google.com/p/openpgm>) project. It is supported on a large set of OS (including Linux, Solaris, Mac OS X and Windows). The library uses the LGPL license while the given tools are GPL.

The following lines are two questions / answers taken from the ZMQ FAQ:

How come ØMQ has higher throughput than TCP although it's built on top of TCP?

Avoiding redundant networking stack traversals can improve throughput significantly. In other words, sending two messages down the networking stack in one go takes much less time than sending each of them separately. This technique is known as message batching.

When sending messages in batches you have to wait for the last one to send the whole batch.

This would make the latency of the first message in the batch much worse, wouldn't it?

ØMQ batches messages in opportunistic manner. Rather than waiting for a predefined number of messages and/or predefined time interval, it sends all the messages available at the moment in one go. Imagine the network interface card is busy sending data. Once it is ready to send more data it asks ØMQ for new messages. ØMQ sends all the messages available at the moment. Does it harm the latency of the first message in the batch? No. The message won't be sent earlier anyway because the networking card was busy. On the contrary, latency of subsequent messages will be improved because sending single batch to the card is faster than sending lot of small messages. On the other hand, if network card isn't busy, the message is sent straight away without waiting for following messages. Thus it'll have the best possible latency.

5.2 ZMQ and multicasting

Even if the PGM protocol used by ZMQ when doing multicasting is a reliable multicast protocol, it seems that it is not "that" reliable and has some traps in which it is easy to fall down. The author was not able to build a reliable connection between 2 computers in the ESRF network using the openPGM embedded with ZMQ. There was always some packets lost during the re-transmission. Therefore, all the test done in the following sub-chapters were done using the **publish - subscribe messaging patterns but with the TCP protocol** (therefore with as many copies on the network as the number of clients)

5.3 Performance

ZMQ only transport data and do not provide any encoding/de-coding system. For the following test, the data exchanged between the client(s) and the server are exactly the data sent by a Tango event. We have re-used the omniORB marshalling and un-marsahlling system to encode/decode the data. This means that the data within the ZMQ message are coded using the CORBA CDR (Common Data Representation) and are exactly the same data than those transferred by a Tango event.

5.3.1 Using remote hosts

The client is running on the host called *pcantares* and the device server processes run on *eclipse*. The tests were done with a ZMQ sender defining a ZMQ socket with a High Water Mark of 1 and a Low Water mark of 0. The results are the following:

Clnt	1 DevLong				1024 DevLong				262144 DevLong			
	T. V7	Group	DDS	ZMQ	T. V7	Group	DDS	ZMQ	T. V7	Group	DDS	ZMQ
1	770	1330	12500	45000	650	800	1850	2400	10.7	8.5	7.9	10.7
2	770	1320	10500	27000	460	570	1800	1200	5	4.3	7.9	4.7
5	400	870	7900	14000	200	280	1800	500	2.3	1.7	8	1.8
10	220	450	6500	7300	100	150	1700	230	0.9	0.8	8.1	0.8

From this table, it is clear that we have an increase in term of performances when using small amount of data (Attribute of 1 DevLong) even compared to DDS which is using multicasting. For medium size data attributes, the increase of performance using ZMQ compared to DDS is significant only in case of a single client. As soon as you increase the number of clients, DDS is faster (multicasting effect). For large data transfer, 3 solutions (including ZMQ) gives nearly the same results while DDS performs better if you increase client number (multicasting).

5.3.2 Device server and clients on the same host

The computer used for this test is *pcantares*. Tango 7.1.1 has been used for the Group case. The results are:

Clnt	1 DevLong				1024 DevLong				262144 DevLong			
	T. V7	Group	DDS	ZMQ	T. V7	Group	DDS	ZMQ	T. V7	Group	DDS	ZMQ
1	620	2500	5300	16000	600	1830	4500	15000	40	33	91	100
2	420	1250	3200	9800	400	930	3200	7300	30	16	35	60
5	230	520	1500	3900	210	380	1350	2700	15	7	32	26
10	170	250	770	1600	160	190	700	1200	8.5	3.3	17	13.5

From this table, it is clear that we have an increase in term of performances with ZMQ.

5.4 ZMQ and Java

ZMQ has a Java binding. This is not pure Java but a java interface to the C library. This means that java application using ZMQ depends on the presence of a library on the computer. This modify the portability of Java application.

5.5 Implementation in Tango events

The information exchanged between a CORBA client and a server when the client requests the server to execute a IDL operation are:

- A GIOP header including the operation to be executed
- The object key on which the operation has to be executed

- The data themselves coded using the CORBA Common Data Representation (CDR)

In the case of Tango event, we don't need any object key because there is only one object involved in the event receiver. We also do not need the remaining GIOP header. Therefore, we could simplify the data transferred to:

1. The operation to be executed (coded using one integer)
2. The data themselves coded in CDR

This has been partly experimented during the performance testing of ZMQ where the data sent on the network were encoded/decoded using omniORB generated methods for CDR encoding/decoding. The operations that we need to implement are exactly the same than those defined for the Tango group solution (5 operations requested).

A Tango client interested in events, instead of being a CORBA server will be a ZMQ subscriber. The Tango server instead of being a Tango client will be a ZMQ publisher.

We have to check if it is also possible to decode a CDR encoded data buffer with JacORB for Java application receiving Tango event.

6 Compatibility with previous Tango event system

TODO

7 Conclusion

If we try to summarize the pros and cons of each solution:

- Group
 - Pros: Lightweight (no extra process, only clients and servers), Tango everywhere
 - Cons: Performance, Amount of code to be written
- DDS
 - Pros: Performance, features, not so many code to be written
 - Cons: Heavy (3 processes and a shared memory), Data partitioning impossible, Integration into Tango (the XML configuration file), difficult configuration (shared memory size, communication buffer size tuning), not easy to use in every day life (kill -9 forbidden), Java portability
- ZMQ
 - Pros: Performance, lightweight (only client and server processes)
 - Cons: Amount of code to be written, Java portability, Young product