Exchanging data distribution layer

Kennet Rode Nielsen
Computer Science, Aarhus University
Aabogade 34, 8200 Aarhus N, Denmark
kennetn@gmail.com

June 2018

Abstract
In this report I investigate if it is possible to replace a custom-made data distribution layer (DDL) with an off the shelf solution. I will investigate the performance implications in a scenario that will stress the DDL. The replacement I will investigate is a data distribution service (DDS) provided by a company called Real-Time Innovations (RTI).
# Table of contents

1 Motivation .......................................................................................................................... 3

2 Problem statement ............................................................................................................... 4
   2.1 Hypothesis ...................................................................................................................... 5
   2.2 Delimitation of scope ..................................................................................................... 5
   2.3 DDS compared to other messaging systems ................................................................. 6
   2.4 Terminology .................................................................................................................. 7

3 Method .................................................................................................................................. 8
   3.1 Hypothesis 1 ................................................................................................................... 8
   3.2 Hypothesis 2 ................................................................................................................... 9

4 Analysis and Results ............................................................................................................. 11
   4.1 Architectural reconstruction of the DDL ....................................................................... 11
   4.2 Selecting DDS provider ................................................................................................ 15
   4.3 Finding adaptor requirements ...................................................................................... 19
   4.4 Implementation ............................................................................................................. 26
   4.5 Performance test .......................................................................................................... 31
   4.6 Identify Test Environment ............................................................................................ 32
   4.7 Identify Performance Acceptance Criteria ................................................................. 33
   4.8 Plan and Design Tests ................................................................................................... 34
   4.9 Configure Test Environment ....................................................................................... 44
   4.10 Implement Test Design ............................................................................................... 44
   4.11 Execute Tests .............................................................................................................. 45
   4.12 Analyze, Report and Retest ......................................................................................... 45

5 Related work ......................................................................................................................... 58
   5.1 Evaluating the Performance of DDS ........................................................................... 58
   5.2 Discovery Protocol for DDS ....................................................................................... 58

6 Conclusion .......................................................................................................................... 59
   6.1 Hypothesis 1 ................................................................................................................... 59
   6.2 Hypothesis 2 ................................................................................................................... 59

7 List of figures ....................................................................................................................... 61

8 References ........................................................................................................................... 62
1 Motivation

In my company we are developing systems that control airfield lighting on airports. The systems are used by the air traffic controllers to monitor the status of the lights and change the brightness. We use standard Windows PC’s distributed across the airport to facilitate this. Each system is composed of many applications that communicate. It is essential that the information displayed to the air traffic controllers is up to date. If the system cannot display up to date information it must show that the information is unavailable. The systems all use a proprietary messaging system called DDL. The DDL allows applications to exchange messages regarding state changes via central servers. The messages are exchanged via topics. The DDL was designed in the 90'es and was developed without unit tests or compliance to international standards for software development. A simplified version of a system that use the DDL is shown in Figure 1.

The figure shows how the different applications connect to a DDLClien on the same workstation and how this DDLClien connects to redundant DDLServers in a central location. If these central servers are unavailable communication between applications on different workstation will not be possible.

We have been asked to provide a more decentralized system architecture and for now we have not been able to deliver this. The Wikipedia description for decentralized system is

Wikipedia
A decentralised system in systems theory is a system in which lower level components operate on local information to accomplish global goals.
Throughout this report I will define a decentralized system as a system where workstations in one location can continue to operate on the local information despite loss of communication to the other locations.

The reason the customers want to have this is to create bulkheads [Nygard: 5.3 Bulkheads] between partitions of the system. In Figure 1 each location is a partition of the system. The Human Machine Interface (HMI) Application in Location 1 will have many topics that is exchanged with the IO Application in the same location 1. Together the IO and HMI application can provide partial system functionality. Therefore, eliminating the need to communicate via central servers will reduce the risk of total system failure.

For the reasons mentioned above I would like to investigate if it is possible to replace the messaging system with an of the shelf solution that complies to international standards for software development and that provides a decentralized system architecture.

## 2 Problem statement

The new solution will have to replace the DDL without requiring that the applications that use the DDL is changed. This is because we have many applications that use the DDL, making it very time consuming to rewrite them all.

All the applications that use the DDL use the same API. This API is implemented in a Dynamic Link Library (DLL) that they all include, so a replacement will have to provide the same API. It will of cause not be possible to find a solution that has the same API as the DDL, so some adapter design pattern [Christensen] will be required. The adapter should be simple so that I am not creating a new DDL on top of another messaging system. Because of this a messaging system with similar properties to the DDL is preferable.

Initial investigations have shown that Data Distribution Service (DDS)[DDS] looks like a good alternative to DDL. DDS is a middleware protocol and API standard for data-centric connectivity from the Object Management Group (OMG). The standard describes both a wire protocol [DDSProt] and an API interface [DDSSpec]. This makes it possible to easily change DDS provider in a system and to connect DDS products from different vendors to each other.

The investigations have also shown that RTI has a DDS solution that is implemented according to the international standards for software development that we would like to comply to. In a DDS solution all the applications can communicate directly to each other in a peer-to-peer style without the need for a central server. In Figure 2 the same applications as Figure 1 is shown but in a DDS architecture.
In the architecture there is no longer a central location with a server this makes the architecture decentralized because the HMI application and the IO application can communicate despite the network to Location 2 being unavailable. Because the adaptor will replace the DDL it will have to provide the same performance as the DDL to ensure that the user experience is not degraded after an update. To confirm this, I will have to investigate the performance implications when exchanging the DDL with a DDS solution. It will not be feasible to investigate the performance implications in every way the DDL is used. Therefore, I will instead look for a scenario that stress the DDL to see how this will perform when implemented via a DDS adaptor.

2.1 Hypothesis
I will investigate the following hypotheses (HS)

| HS 1 | It is possible to create a DDS adaptor with less than 1000 lines of code for the DDL that will have publish, subscribe and update functionality. |
| HS 2 | A DDS solution via an adaptor pattern can provide the same or better topic update performance in a scenario that will stress the DDL. |

2.2 Delimitation of scope
The current DDL solution has some features that are unusual for a messaging system. When evaluating the hypothesis, I will not take those features into account. Instead the adapter will be returning dummy objects and fake statuses when these features are used. The hypothesis is about the publish, subscribe and update functions of the DDL.
One of the reasons for changing is to find a solution that is implemented according to international standards for software development, however support for these advanced features is not supported by community (no charge) editions and because this is just a pre-study I do not have the option to buy a DDS solution. Therefore, when selecting a DDS provider, I will only look at options that are free.
2.3 DDS compared to other messaging systems

DDS and other messaging systems are not necessarily competing technologies, each has advantages and disadvantages. It is like comparing SQL and NoSQL databases. Choosing the right tool for the right job is important.

DDS uses a peer-to-peer system architecture, this should scale better horizontally because adding topics will only affect other applications until the subscribers and publishers have found each other. In the DDL and other messaging systems like RabbitMQ, and ActiveMQ there is a server that will have to relay the additional messages. RabbitMQ does have some horizontal scaling options but this is not a core feature.

In a DDS the collection of all topics published is called the global data space and can be thought of as a form of distributed shared memory or distributed data store. The global data space has similarities with a tuple space. However, in a tuple space the tuples are accessed via contents while in the Global data space the data is accessed via a topic name.

As proven by Eric Brewer's CAP theorem you can only have two of the three in Consistency, Availability and Partition tolerance. In the global data space availability to chosen over consistency, but it does have eventual consistency.

Another feature of DDS is the data type for every topic. When a topic is published each part of the data transferred must be described. It is easy to think of this as publishing a class and sending instances of the class. This ensures good performance because the data will always be transferred in binary format. In systems like RabbitMQ, ActiveMQ, SwiftMQ the marshalling is done in the application code so it would be possible to achieve the same by for instance always sending json encoded data, but that would require additional bandwidth for every message.
2.4 Terminology

In this report I will be creating an adaptor between two different middleware solutions. These solutions use different terminology. In Table 1 the term used in each middleware solution is listed along with a short description of the term.

<table>
<thead>
<tr>
<th>DDS term</th>
<th>DDL term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>Item</td>
<td>A named stream of data</td>
</tr>
<tr>
<td>Sample</td>
<td>Value</td>
<td>An instance of data on a topic that a publisher sends to a subscriber</td>
</tr>
<tr>
<td>Topic type/</td>
<td>ObjectMap</td>
<td>Description of data on a topic</td>
</tr>
<tr>
<td>TypeCode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primitive type</td>
<td>Type</td>
<td>The basic programming language types like integer and boolean.</td>
</tr>
<tr>
<td>TypeCode</td>
<td>SubItem</td>
<td>Element of a Topic type that has a name and a primitive type.</td>
</tr>
<tr>
<td>Participant</td>
<td>DDLSubClient</td>
<td>The API class in the messaging system used to publish and subscribe.</td>
</tr>
</tbody>
</table>

I will primarily use the DDS terminology in the report. However, I will not translate functions and names used in source code therefore diagrams and descriptions of DDL will use DDL terminology.
3 Method

3.1 Hypothesis 1
To confirm or reject hypothesis 1 I need to build an adapter that will bridge between DDL and a specific DDS provider. This task can be split into 4 steps that is described below.

3.1.1 Understand DDL
I need to understand the current DDL API to know how the interface works and how it can be mapped to the DDS API. To do this I will do an architectural reconstruction as described in the Symphony process [Symphony].
As described in Symphony I will only create views of the source that is needed to solve the problem elicitation. The adaptor will only implement the publish and subscribe functions of the DDL therefore I only need views of these. Because I do not want to modify existing applications it is essential that I do not only look at the two methods but the entire sequence when publishing and subscribing. Therefore, I have determined that my target views should show this part of the interface. The viewpoints I will use to show this is UML sequence diagrams [UML]. I use sequence diagrams because they show methods as well as the sequence in which they are used and this is what the adaptor needs to emulate. Because the architecture will be changed to a decentralized system architecture it is also important to understand the components that will be removed and what functions they performed. For this I will create a UML Deployment diagram.

3.1.2 Select DDS provider
I need to select a DDS provider that I can use to implement the adapter towards. I will evaluate the different providers based on price and if the license allows me to use the software for the duration of the project. I want to select a DDS provider that is maintained, therefore I will look at what version of the DDS standard the provider complies to.

3.1.3 Finding adaptor requirements
To find the requirements for the adaptor I first need to understand the DDS API. I will use examples and online documentation to understand how the DDS API is intended to be used. In case this is insufficient there are also online forums that can be used to ask questions. Based on this I will create UML diagrams that show how the API should be used. This will allow me to easily compare the two APIs and see what the adaptor needs to do to bridge between them.

3.1.4 Implementation
When implementing the adaptor, I will use the information gathered from the UML diagrams. But the implementation will be an iterative process where I gradually get a better understanding of the DDS API so the actual adaptor implementation will not be as the online examples.
3.2 Hypothesis 2

After I have implemented the adaptor with the DDL API I can compare the performance of the DDS adaptor with the DDL. For this I will need to do performance tests on both messaging systems. The result of these tests can be compared to see if the DDS can provide the same or better performance as the DDL in a stress scenario.

Microsoft has defined seven core performance testing activities for web applications these are listed below [MSPref]. Despite the list originally being made for web applications I see no reason that it should not work for the tests I will be doing.

To confirm or reject hypothesis 2, I will use this list to identify activities that I need to perform.

3.2.1 Identify Test Environment
This activity is about understanding the difference between the environment where the test is executed and the real environment. The ideal test environment is a system that is identical to the real environment where the DDL is used.

3.2.2 Identify Performance Acceptance Criteria
This activity is about defining when a system has sufficient performance. It will always be possible to improve performance, but this will have a price. Therefore finding what is the acceptable performance level for the customer is important. Having this defined clearly makes it easy to know when to stop improving performance.

3.2.3 Plan and Design Tests
This activity is about defining multiple user interactions with the system to measure performance on. These interactions are called a workload model, a workload model should be realistic and contain key usage scenarios. The different user interactions should be mixed together to create a realistic model of how the system is used by multiple users at the same time. Running the workload model on a web system is a good way to measure its performance because it creates a realistic environment with many simultaneous user interactions. This can for instance show if some scenarios in the system is affecting other scenarios. This would not be visible if performance was measured on one scenario at a time.

3.2.4 Configure Test Environment
This activity is about setting up the tools used when the test is executed. The tools include hardware, operating system and additional software.

3.2.5 Implement Test Design
When implementing the test, it is common to use tools that simulate the users by sending mouse clicks towards the GUI of the system. In a web system this would be a browser showing a web page. However, web tests can also be implemented by having a program directly sending HTTP requests towards the web server. There are many ways to implement the tests.
3.2.6 Execute Tests
This activity is the actual running of the test. For this it is important to consider how to best make the test results reproducible. It should also be considered what parts need to be monitored to ensure that the test is executed correctly.

3.2.7 Analyze, Report and Retest
The final activity is processing the results from the test into something that will provide good insight into the performance of the system. For this I will use charts to show the difference between the DDS and DDL. I will also try to improve the performance in case the acceptance criteria are not met and afterwards run the tests again.
4 Analysis and Results

4.1 Architectural reconstruction of the DDL

The first step in the architectural reconstruction of the DDL is to understand the software components used. To gather the information for the UML Deployment view shown in Figure 1, I have used my years of working with the DDL. Figure 1 shows four component types DDLServer, DDLClient, DDLSubClient and Application that make up a system. The application component is a simplification of many different applications that perform different functions. In both Figure 1 and Figure 2 a IO application and HMI application is shown and I will be using these as examples of applications that publish a topic and subscribe a topic throughout the report. Each application will load a DLL that contains the functions needed to publish and subscribe topics. The part of the DDL that runs in each application is called the DDLSubClient.

4.1.1 Establish communication

Before I look at the publish and subscribe functions I will shortly describe how the link between publisher and subscriber is established, this is important because it will be different in the DDS. Figure 3 shows how communication is established between DDLServer, DDLClient, and DDLSubClient. The sequence shows that a DDLClient is started and afterwards an application using the DDLSubClient is started. The DDLServer is running initially.

When the DDLClient is started it will try to connect to the DDLServer via a TCP/IP connection. It uses a host name and port number read from a configuration file when doing this. In the diagram it is shown that it sends a TCP/IP connect request and that the DDLServer responds
that they can connect. When the IO application starts it creates a DDLSubClient and calls the PublishItem method. The details of the PublishItem is described later. Once the DDLSubClient is created it will try to connect to the DDLClient with a TCP/IP connection to localhost. When the DDLSubClient connects it will send a list of all topics published and a list of topics subscribed to the DDLClient. These lists contain only information about topics that should be exchanged, there is no actual values exchanged. When the DDLClient receives the lists, it filters out the topics that it may already have send to the DDLServer and forwards the new ones. The DDLServer can now match published topics and subscribed topics and start sending values. The DDLServer stores the latest value so that when a subscriber is added it gets the latest value the server received. The two main functions of the DDLSubClient API is to Publish and Subscribe topics. Below I will describe each of these.

4.1.2 Publish API
When an application wants to publish information for instance the IO application may read a temperature value and pressure value from some hardware, then it creates a topic where it can publish this information. Before the application creates the topic, it will dynamically describe the topic contents, this description is called a topic type. A Topic is called DDLItem or just item in the DDL. In the IO application example there is one topic type with two members called temperature and pressure and both members are double values. First it creates a DDLObjectMap and populates it by calling AddSubItem with the type of each SubItem, once it is finished it sends the map to the DDLSubClient via the PublishItem function along with a topic name. This is shown in Figure 4.

![Figure 4 DDL Publish](image-url)
The IO application example would look as shown below in code.

```cpp
DDLObjectMap map;
map.AddSubItem( "temperature" , CObject::eDOUBLE );
map.AddSubItem( "pressure" , CObject::eDOUBLE );
item1 = DdlSubClient->PublishItem( "Sensor1" , map );
```

The API method the application calls is PublishItem but this will not send any values to the subscribers, it is only the topic type that is published. The PublishItem returns an item that is used to set the value to be published. This can be seen in Figure 5.

The Application calls SetItemData for each SubItem that was added to the DDLObjectMap. The application must ensure that it calls SetItemData with the right types that was specified in the DDLObjectMap. Once it has done this it sends out the update by calling UpdateDDLItem. In the IO application example it would look like this.

```cpp
item1->SetItemData( "temperature" , 22.5 );
item1->SetItemData( "pressure" , 1.013 );
DdlSubClient->UpdateDDLItem( item1 ); // Send out update
```

After this all subscribers of the topic are notified asynchronous if there is a connection between the components in the system.

### 4.1.3 Subscribe API

The applications subscribe to topics by using the DDLSubClient. They call SubscribeItem with the name of the topic and a callback function that should be called when the topic is updated. The DDLSubClient creates a DDLItem and returns it to the application so that it can read the topic at any time. This is shown in Figure 6
An example of this could be a HMI application that wants to display the temperature from the IO application. The code for subscribing would look like this.

```cpp
item = DdlSubClient->SubscribeItem( "Sensor1", DLLCallBack );
```

Figure 7 shows the sequence when a publisher has updated a topic and the DDLSubClient calls the function supplied when the topic was subscribed.

When the application is notified it must call the function IsValid to know if the update is because of a new value or because the connection with the topic publisher is unavailable. In case a new value is received the application can read the value by calling GetItemData with the name of the SubItem it wants to read.

In the HMI application example the callback function would look like this.

```cpp
void Test::DLLCallBack( DDLItem* item )
{
    if( item->IsValid() )
    {
        double temp;
        if( item->GetItemData( "temperature", temp ) )
        {
            // use the temperature value
        }
    }
}
```
In the example the SubItem name temperature and topic name Sensor1 is hardcoded. However, the HMI application would normally read both of those from a configuration file. This makes it easy to change the values a HMI shows because the configured topic can just be changed. If a new IO application with different SubItem names is created the HMI can also show values from this without changing the code.

### 4.1.4 Adapter classes

From the above diagrams it is possible to see the minimum classes and methods the Adapter needs to implement and map to DDS equivalent methods. These are shown in Figure 8.

*Figure 8 DDL API Classes*

The DDLItem class has additional Get and Set methods with other types but these are not shown to keep the diagram simple.

### 4.2 Selecting DDS provider

To implement the adaptor, I need to select a DDS provider that I can use to implement the adapter towards. The DDS standard is composed of three core standards [DDSCore]. There is the Data-Centric Publish-Subscribe (DCPS) model, the Real-Time Publish-Subscribe wire protocol (RTPS) and the Interface Definition Language (IDL).

The RTPS specification ensures that products from different vendors will work together. The DCPS model ensures that the API is similar across different vendors. The DCPS part of the DDS standard is commonly referred to as DDS specification, but to avoid confusion I will use the term DCPS specification in this report.

The Interface Definition Language standard describes a neutral language used to describe topics. When an application wants to publish a topic the developer will create a IDL file that contains a description of the topic type. This IDL file is converted into source code of the selected programming language by a tool supplied by the DDS provider. The source code can then be included into the workspace of the application and compiled. This creates a class that the
developer can create an instance of when updating a topic. This makes it easy to update a topic because the developer will work directly with a class instance that can be send to the DDS.

From the architectural reconstruction we can see that the DDL describes each topic at runtime. This creates a problem for the adaptor because it will not be possible to create a IDL file. We can solve this by extending the functionality of the adaptor or we can use the XTypes extension. The XTypes extension allows for dynamic data definition at runtime in DDS.

Solving the problem by extending the functionality of the adaptor will add more code to the adaptor and this is not desirable because we want to keep the adaptor as simple as possible to have the best chance to confirm Hypothesis 1. The XTypes extension is however an optional extension that the providers may choose to implement. Therefore when selecting a DDS provider, I will investigate if they support XTypes and what version it is. DDS Providers with XTypes support will therefore be preferred over other providers.

I want to select a DDS provider that is maintained, therefore I will look at what version of the DDS standard the provider complies to. The latest versions of the standard published by OMG is listed in Table 2.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Version</th>
<th>Published date</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPS</td>
<td>1.4</td>
<td>April 2015</td>
</tr>
<tr>
<td>RTPS</td>
<td>2.2</td>
<td>September 2014</td>
</tr>
<tr>
<td>XTypes</td>
<td>1.2</td>
<td>August 2017</td>
</tr>
</tbody>
</table>

Table 2 Latest DDS specifications

When investigating the providers, I found that not every provider has published clear information on their websites, so I decided not to spend more than 2 hours for each provider searching for details on their products.

From the OMG DDS website [DDS] I found several providers that I have investigated. I have listed the products in the next chapters with a short description of the information I found about each product.

### 4.2.1 OpenDDS

OpenDDS [OpenDDS] is an open source implementation of DDS that can be found on GitHub. The source code was developed by a company called Object Computing, Incorporated. The company maintains the code on GitHub that can be used commercially without any fees. Object Computing, Incorporated provides Commercial Support and training. There is no indication that OpenDDS supports XTypes on the website and forum posts also suggests it is not supported [OpenDDSXTYPE].
4.2.2 Vortex OpenSplice

OpenSplice [OpenSplice] is a commercial solution that also has a Community Edition. The commercial solution has additional tools, protocols and services in addition to the core functionality. It was made by a company called PrismTech that was later acquired by ADLINK. The Community Edition is licensed under Apache 2.0 and can be used commercially. After e-mailing with ADLINK I found that they are working on adding XTypes but it is not supported currently.

<table>
<thead>
<tr>
<th>Product name</th>
<th>OpenSplice Community Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>6.9.0</td>
</tr>
<tr>
<td>Company</td>
<td>ADLINK</td>
</tr>
<tr>
<td>Licence</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
</tr>
<tr>
<td>DCPS Specification</td>
<td>1.4</td>
</tr>
<tr>
<td>RTPS Specification</td>
<td>2.2</td>
</tr>
<tr>
<td>XTypes</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

4.2.3 Connext DDS

Connext DDS [ConnextDDS] is created by RTI. RTI provides 4 different solutions targeting different market segments. One of the solutions is the Connext DDS Cert that includes Reusable certification evidence for IEC 61508 SIL 3 to help certify the final product. IEC 61508 is one of the standards for software development we have been asked to comply to. RTI has a University Licences that allows students to use the Connext DDS Professional solution in research like this. Connext DDS Professional has partial support for XTypes [ConnextDDSXtype]

<table>
<thead>
<tr>
<th>Product name</th>
<th>Connext DDS Professional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>5.3.0</td>
</tr>
<tr>
<td>Company</td>
<td>RTI</td>
</tr>
<tr>
<td>Licence</td>
<td>University Licences</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
</tr>
<tr>
<td>DCPS Specification</td>
<td>1.4</td>
</tr>
<tr>
<td>RTPS Specification</td>
<td>2.2</td>
</tr>
<tr>
<td>XTypes</td>
<td>1.1 (Partly)</td>
</tr>
</tbody>
</table>
4.2.4 CoreDX DDS
CoreDX DDS [CoreDXDDS] is created by Twin Oaks Computing, Inc. The web site is limited with details about the product. I was not able to find any information on what specifications they comply to. They do provide University Licenses that this project should be eligible for, however I did not confirm this because this is decided by Twin Oaks on a case by case evaluation and I did not want to ask for a licenses that I did not intend to use.

<table>
<thead>
<tr>
<th>Product name</th>
<th>CoreDX DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4</td>
</tr>
<tr>
<td>Company</td>
<td>Twin Oaks Computing, Inc.</td>
</tr>
<tr>
<td>Licence</td>
<td>University Licences</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
</tr>
<tr>
<td>DCPS Specification</td>
<td>?</td>
</tr>
<tr>
<td>RTPS Specification</td>
<td>?</td>
</tr>
<tr>
<td>XTypes</td>
<td>?</td>
</tr>
</tbody>
</table>

4.2.5 eProsima Fast RTPS
eProsima Fast RTPS [eProsima] is made by eProsima. The software is licensed under Apache 2.0 and eProsima provides commercial support. I found no information about the DCPS Specification used. I did find some slides [eProsimaXtype] that indicate support for XTypes but there was no version reference so I cannot confirm what version it may be.

<table>
<thead>
<tr>
<th>Product name</th>
<th>eProsima Fast RTPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1.5.0</td>
</tr>
<tr>
<td>Company</td>
<td>eProsima</td>
</tr>
<tr>
<td>Licence</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
</tr>
<tr>
<td>DCPS Specification</td>
<td>?</td>
</tr>
<tr>
<td>RTPS Specification</td>
<td>2.2</td>
</tr>
<tr>
<td>XTypes</td>
<td>?</td>
</tr>
</tbody>
</table>

4.2.6 Decision
After having evaluated the different providers I decided to get a University Licences from RTI for Connext DDS Professional. I did this because.

1. They have partial XTypes support
2. They can provide certification evidence for IEC 61508 SIL 3.
3. They support the latest version of DCPS and RTPS.
4. Their web site has a user forum with few unanswered questions giving me confidence that I can get help.
4.3 Finding adaptor requirements

To find the requirements for the adaptor I need to understand the DDS API. I will need to understand the publish and subscribe part of the interface and how this looks compared to the DDL. For each of the functions I have created UML diagrams to compare with the DDL counterpart and identified requirements for the adaptor.

Initially I will look at how communication is established in the DDS to find requirements that may arise because this is different than how the DDL establishes communication. Because the DDLServer and DDLClient will be removed any functionality that they provided will also have to be replaced.

4.3.1 Establish communication

The RTPS wire protocol is divided in two parts, the Behavior Module and the Discovery Module. The Behavior Module describes how data is exchanged between a publisher and subscriber after the connection has been established. The Discovery Module handles identifying the applications in the network and setting up the connections. The standard specifies that the Discovery Module must support at least the Simple Participant Discovery Protocol (SPDP). In addition to this DDS providers may also implement additional protocols that provide better performance or other features. The default RTI configuration for SPDP will use IP multicast [RFC1112] to discover other applications. SPDP specifies that the default multicast address to use should be 239.255.0.1. This address is an administratively scoped address in the (IPv4 local scope) range. Routers will not send this address to the WAN side of the router, so only applications within the same LAN will be able to multicast to each other.

Figure 9 shows how two applications use SPDP and multicast to find each other and exchange publish and subscribe information.

![Figure 9 DDS Connecting](image-url)

In the figure the IO application starts and joins a multicast group. It does this by sending a message to the router that informs the router to send all multicast addressed to 239.255.0.1 to the
application. Afterwards it will announce its own presence by sending to the multicast address. At this point there is no other applications that have joined the multicast so the message stops in the router. When the HMI application is started it does the same but now the announce message will be send to the IO application. This triggers the IO application to send publish and subscribe information to the HMI application and this then triggers the HMI application to do the same.

For the adaptor the use of multicast is a good solution because the applications will not need to have a configuration with the IP address of the other workstations in the system.

<table>
<thead>
<tr>
<th>Adaptor implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS will establish communication without any configuration input from the adaptor.</td>
</tr>
</tbody>
</table>

One drawback of the multicast solution is that it requires support in the network routers. Another drawback is that when multiple systems that should not connect to each other is using the same local network it is more likely that they may accidentally connect to each other compared to a solution where the host addresses is configured in each system. This will however not be an issue for the adaptor or in real systems, because there will only be one DDS system running.

The Behavior Module that handles communication after the applications have discovered each other can be configured in different ways. This configuration can change many aspects of the communication for instance how samples are exchanged and how many historical samples a publisher will keep. The Adaptor will need to implement the same behaviour as the DDL. The DDL uses TCP/IP for communication this ensures that even if packets are lost in the network the sample published will be delivered to the subscriber eventually. The adaptor will not have to use TCP/IP but it will have to provide the eventual consistency. All subscribers must receive the same latest sample in a system where all connections are operational.

<table>
<thead>
<tr>
<th>Adaptor requirement 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>All subscribers must receive the same latest sample</td>
</tr>
</tbody>
</table>

In the DDL the server stores the latest sample so that when a new subscriber is added it gets the latest sample the server received. In the DDS there is no server therefore this storing of the latest sample must be handled in the adaptor or the DDS participant must be configured to handle this.

<table>
<thead>
<tr>
<th>Adaptor requirement 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>New subscriber must receive latest published sample</td>
</tr>
</tbody>
</table>

4.3.2 Publish API
From the architectural reconstruction I found that the DDL describes each topic at runtime and when I selected DDS provider I selected a provider with XType support to be able to describe the topic type at runtime.
To see how to use XType in RTI’s DDS I searched their website [RtiDynDataApi] and the forums [RtiForumDyn]. There I found that it is possible to dynamically describe the topic type with the TypeCodes class. In Figure 10 it is shown how the application first creates a TypeCode instance via a type code Factory and then adds members. Once the members have been added the TypeCode is registered with a type name.

For the IO application example shown under DDL Publish the code would look as shown below. Some function parameters and error check are not shown to keep the example simple.

```
DDS_TypeCode * type = factory->create_struct_tc( "SensorType1" );
type->add_member( "temperature", factory->get_primitive_tc(DDS_TK_DOUBLE) );
type->add_member( "pressure", factory->get_primitive_tc(DDS_TK_DOUBLE) );
dyn_data_type = new DDSDynamicDataTypeSupport( type );
dyn_data_type->register_type( participant, "SensorType1" );
```

The sequence is very similar to the first part of the DDL publish sequence seen in Figure 4. One difference is that when add_member is called it is not just a primitive type but a TypeCode that must be passed. This is because the DDS supports complex structures with TypeCodes that contain TypeCodes. Therefore TypeCode could be used to describe both the topic type and the members of the topic type. To avoid confusion, I will refrain from using TypeCode when describing the topic type and only use TypeCode when describing topic type members.

For the adaptor I will only need primitive types and the TypeCode for these are retrieved via the getPrimitive_tc method. The adaptor will need to map from the DDL primitive types to DDS primitive types.

**Adaptor requirement 3**

Map DDL primitive types to DDS primitive types

Another difference is that the topic type must be registered with a unique type name before it can be used to create a topic. The DDL does not have any names for the types that are created, so the adaptor must create a name dynamically.
Adaptor requirement 4
Generate a unique type name based on the dynamic type description

Once the type name has been registered the topic can be created by using the DDSDomainParticipant::create_topic as can be seen in Figure 11.

For the IO application example the code would look as shown here.

```cpp
topic = participant->create_topic( "Sensor1", "SensorType1" );
DDSDataWriter * writer = publisher->create_datawriter( topic );
dyn_writer = DDSDynamicDataWriter::narrow( writer );
```

After the topic is created a DataWriter is created for the topic to be able to update it. This is done via the DDSPublisher that returns a DataWriter that can be converted into a DDSDynamicDataWriter. Compared to the DDL where this was all handled in the PublishItem method this is a bit more complicated. The reason the DDS interface gets more complicated is because it also supports classes create from IDL files that represent the topic type. When these classes are used instead of the dynamic type creation shown, the code is just as simple as the DDL equivalent and has the advantage of type checking.

When the application wants to update a topic it creates a sample to be published. A sample is a value that a publisher sends to a subscriber. Creating a sample is done by calling create_data on the DDSDynamicDataTypeSupport that was created when the type was created. This is shown in Figure 12.
For the IO application example the code would look as shown here.

```c++
DDS_DynamicData * sample = dyn_data_type->create_data();
sample->set_double( "temperature", 22.5 );
sample->set_double( "pressure", 1.013 );
dyn_writer->write( sample ); // Send out update
```

The use of create_data allows the application to have different samples it can work with before calling the write method with one of them. For the adapter this will not be necessary, it will only need to store a single sample because the DDL did not have that feature.

**Adaptor requirement 5**

Store a single DDS sample to send to DataWriter when UpdateItem is called.

The members of the sample are updated by calling set methods, and just like the DDL the application needs to ensure it uses the right primitive type when calling the set methods. After the members are updated the sample is send out by using the DDSDynamicDataWriter that was created when the topic was published.

### 4.3.3 Subscribe API

When an application subscribes a topic in DDS the initial steps are the same as when a topic is published. First the topic type must be registered as seen in Figure 10 and to register a type the type members must be known. When the topic type is registered the topic can be created. The DDS will then try to match the published topic type with the subscribed topic type to ensure that the subscriber only uses members that are available. This is different than the DDL where the type members used are not checked until the application reads them. In case the members are not in the published topic the GetItemData method in the DDL will just fail. Because of the way the DDL works the adaptor will not be able to know what type members will be used at the time the topic is subscribed. The adaptor will have to find a way to register a topic type that will match the published topic type.
Adaptor requirement 6
Find topic type from topic name and register topic type.

Once the topic type is registered the adaptor can subscribe as shown in Figure 13.

Going back to the HMI application example from DDL Subscribe the code would look as shown below. The code for registering the topic type is the same as shown for DDS Publish.

```cpp
topic = participant->create_topic( "Sensor1", "SensorType1" );
DDSDynamicDataReader * reader = publisher->create_datareader( topic, observer );
dyn_reader = DDSDynamicDataReader::narrow( reader );
```

The difference compared to Figure 11 DDS Publish is the create_datareader instead of create_datawriter. The create_datareader method takes a DDSDataReaderListener in addition to the topic. This listener is the observer role in an implementation of the Observer design pattern [Christensen] in the DDS. The pattern is used to send notifications when the topic is updated. The Adaptor will have to call the callback function supplied in the DDL SubscribeItem method when the observer is notified.

Adaptor requirement 7
Store DDL callback functions so they can be called when notified

When the DDS receives an update the observer is notified via a call to on_data_available as seen in Figure 14.
The HMI application example code looks as shown below.

```c++
void Observer::on_data_available( DDSDataReader* reader )
{
    DDS_SampleInfoSeq info;
    DDS_DynamicDataSeq sample;
    dyn_reader->take(sample, info, ...);
    for (int i = 0; i < sample.length(); i++)
    {
        if (info[i].valid_data)
        {
            double temp;
            if(sample[i].get_double(temp, "temperature") )
            {
                // use the temperature value
            }
        }
    }
    dyn_reader->return_loan(sample, info);
}
```

The application creates a DDS_SampleInfoSeq and DDS_DynamicDataSeq to receive the values from the DDSDynamicDataReader via the take method. Each of these can hold multiple values received on the same topic. This is because in the DDS more than one sample may arrive in a single notification and therefore it must loop thru all samples received.

**Adaptor requirement 8**

Loop thru all received samples and call the callback function for each.

The sequence after this is the same as the DDL shown in Figure 7, except the methods have different names. When the application is notified the application calls methods to retrieve the values. Because of this the adaptor will have to store the sample that it is performing a callback.
for so that when the application calls GetItemData the sample value can be returned. The adaptor also needs to store the sample after the notification from the DDS is finished, because the application may read the latest sample even when it was not updated.

### Adaptor requirement 9

Store latest received sample so that GetItemData can return the sample value when called.

## 4.4 Implementation

When implementing the adaptor the requirements listed earlier must be handled. Most of them are not so complicated to implement and do not need explanation. There are a few that are more difficult to handle and I will describe them here.

### 4.4.1 Adaptor requirement 1

When using RTI DDS most of the characteristic of how data is exchanged is controllable from a configuration file. These characteristic is also called Quality of Service (QoS) settings. When the DDS participant is started it will search for a XML configuration file called USER_QOS_PROFILES.xml. In this configuration file it is possible to create multiple profiles with different names that describe different QoS settings. In the code it is then possible to select a QoS profile by the name. For the adaptor I decided to use a command line option to control what profile name the adaptor should use in the configuration file. This will make it easy to change between different profiles.

The profiles can use inheritance so that one profile can inherit the QoS settings from another and change some of the QoS settings.

The RTI DDS comes will a large set of profiles that are built in [RtiProfiles] and for the adaptor I decided to use one of these as a base for the first adaptor profile. I decided to use one of the most basic profiles in order keep the settings I need to investigate to a minimum to handle requirement 1.

### Adaptor requirement 1

All subscribers must receive the same latest sample

By looking at other profiles I found that the QoS setting I needed to use was RELIABLE_RELIABILITY_QOS under <reliability> in both the <datawriter> and <datareader> sections.
The configuration file with this setting is shown in DDS Profile 1.

```
<qos_profile name="DDLQos" base_name="BuiltinQosLib::Generic.Common">
  <datawriter_qos>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datawriter_qos>
  <datareader_qos>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datareader_qos>
</qos_profile>
```

DDS Profile 1 Reliable

The file contains first a profile name and a base name. The base name is the built-in profile I decided to inherit from.

The RELIABLE_RELIABILITY_QOS setting will ensure that when a publisher needs to send a sample it will use a reliable communication method. This will ensure that even if a network packet is lost the sample will be retransmitted ensuring that all subscribers receive the same latest sample.

4.4.2 Adaptor requirement 2

The <reliability> setting described in requirement 1 will make the communication to the subscribers that are know when a sample is published reliable. Once the sample is delivered to all subscribers the publisher will discard the sample and thereby making it unavailable to subscribers that join later. To handle requirement 2, I need to change this therefore I examined other built in profiles to see how this could be changed. In the profile called Generic.KeepLastReliable the <kind> under <history> was used to control how many samples a publisher would keep even after the delivery of a sample.

By setting the kind to KEEP_LAST_HISTORY_QOS the publisher will keep the number of samples specified in the <depth> setting.

The complete configuration file used to make the latest sample available to a new subscribers is shown in DDS Profile 2.
4.4.3 Adaptor requirement 4

Adaptor requirement 4
Generate a unique type name based on the dynamic type description

To generate a unique type name one solution could be to simply create a name based on a number that is incremented for every published topic. If we extend the IO application example with an additional sensor that provides the same inputs the UML Object diagram would look as shown in Figure 15

This solution will mean that even though the two sensor topics have the same type members temperature and pressure they would use different type names and each would have its own topic type. The DDS will need to exchange not only the topic name but also the complete topic type description for both topics. This solution will result in many topic types that need to be exchanged and it is very likely that this will have a negative effect on performance. I therefore
decided to find another solution. The next solution I tried was to create the name based on the type members. With the same example as before the object diagram will look as shown in Figure 16.

The member names are simply merged to create a long name. When the second topic is published it will result in the same name “temperature, pressure” and therefore it will not create an additional topic type. The only potential problem with this is that the name may get too long for the DDS to handle. After investigating the limit I found that this was 255 characters and decided this would not be a problem.

4.4.4 Adaptor requirement 6

To solve this, I searched RTI’s forums and found that it is possible to subscribe to build-in-topics that provide information when topics are published in the global data space. This will provide information for any topic that is published regardless if it is subscribed by the participant. This allows the adaptor to see what topic type is used when a topic is published. The adaptor can then subscribe with the topic type that was used when publishing ensuring that the types will match.

The adaptor will have to handle when the publish is done before the subscribe and the other way when the subscribe is done before the publish. In Figure 17 the sequence with publish first is shown here the adaptor first gets the topic type via the on_data_available notification.
When the topic type is received the Adaptor will not know if this topic will be subscribed later. Because the information received on the build-in-topics cannot be retrieved later, the adaptor must continuously collect type information for all topics because they may be subscribed later.

When the application subscribes to the topic the adaptor will know the topic type for the topic and it can use this when setting up the subscribe. Setting up will register the topic type shown as register_type. Then it creates the topic shown as create_topic and sets up the Observer pattern shown as create_datareader. The sequence diagram only shows simplified versions of this. The complete diagrams for this were shown in Figure 10 DDS Register topic type and Figure 13 DDS Subscribe.

For the adaptor to collect information about the topic types I decided to simply create a map from topic name to the topic type. The map is populated every time the DDS sends information about a new publisher. When the application subscribes to a topic the adaptor looks up in the map and if it has already seen the topic it continues as shown above.

If the topic is not in the map when subscribe is called, it will be because the subscribe was done before the publish. This is shown in Figure 18.
When the adaptor does not know the topic type it cannot send the Subscribe to the DDS so it must be queued for later. The adaptor must continue and return a DDL item to the application so that the application can continue. Afterwards the adaptor will have to examine published topics to see if they match the queued topics so that it can setup the subscribe in the DDS. Therefore, until a publish is made the topic will only exist in the adaptor as an queued object waiting for a publisher.

When the topic is published the SetupSubscribe can be made and this will register the type and create the topic.

4.5 **Performance test**

After I have implemented the adaptor I will identify the performance test scenario that I will use to compare the DDS adaptor to the DDL. To identify the performance test, I will go thru the 7 testing activities described in chapter 3.2 Hypothesis 2 and listed below.

1. Identify Test Environment
2. Identify Performance Acceptance Criteria
3. Plan and Design Tests
4. Configure Test Environment
5. Implement Test Design
6. Execute Tests
7. Analyze, Report and Retest

In the next chapters I will go thru them one by one.
4.6 Identify Test Environment

The real systems where the DDL is used is not all identical, the systems vary in number of workstations, workstation hardware and network speed. Therefore it is not possible make one definition of what the real environment is, instead I will define an average real system that I will compare to my test environment.

By analysing the systems delivered in the last 5 years I found that the average size is about 16 workstations in a system and the largest system has 65 workstations. There is an average of 7,6 applications using the DDL on each workstation excluding the DDLClient application.

For the network speed most of the systems has 100 Mbit network so this will be the average system network speed. The workstations are medium priced workstations at the time a system is delivered. Therefore they span between Dual Core 3.0 GHz to Quad core with hyper threading 3.7 GHz. I decided that I will set the average at quad core 3.0 GHz.

The machines have between 4 and 8 gigabytes memory however this is not so interesting because each application on average only use 45 megabytes memory and the largest applications use about 600 megabytes memory. By multiplying the average memory used of 45 megabytes with the average 7,61 applications we see that each workstation only uses 342 megabytes memory for the applications. Therefore when a real system is running it will normally only be using a small portion of the total memory. Each workstation will also run a OS that requires about 1.5 Gigabytes memory. This brings the total to about 2 Gigabytes of memory that is in use on each workstation. I will therefore set the average memory used to 2 gigabytes.

The disk size and speed is not relevant because the DDL and DDS will not use the disk while running.

<table>
<thead>
<tr>
<th>Average system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Used memory</td>
</tr>
<tr>
<td>Network</td>
</tr>
</tbody>
</table>

For the test I have the option to use up to 5 native hypervisors, each of them capable of running up to 10 virtual machines. All the hypervisors have Gigabit network and 32 GByte memory. The CPU’s does vary a bit but they are all Quad core with hyper threading and between 3.4 and 4 GHz.

<table>
<thead>
<tr>
<th>Hypervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Network</td>
</tr>
</tbody>
</table>
If I run 3 workstations on a single hypervisor I will get 15 workstation for the performance test. This will almost match the number of workstations in the test system with the average system. A hypervisor will divide the resources between the virtual machines, this is good when you want to get the most out of the hardware because it is usually not all machines that do work simultaneously. For a stress test or spike test this is however not so good because when the test is executed all the virtual machines will be forced to work at the same time. Because the CPU power in the average system is almost as fast as the CPU power in the hypervisors each workstation will only have about 33% of the CPU processing power in the test system compared to the average system. The network bandwidth will be higher than what is available in an average system. This is summarized in the table below.

<table>
<thead>
<tr>
<th></th>
<th>3 VM per hypervisor</th>
<th>Average system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>15 workstations</td>
<td>16 workstations</td>
</tr>
<tr>
<td>CPU</td>
<td>33%</td>
<td>100%</td>
</tr>
<tr>
<td>Memory</td>
<td>2 Giga bytes</td>
<td>2 Giga bytes</td>
</tr>
<tr>
<td>Network</td>
<td>333 Mbit</td>
<td>100 Mbit</td>
</tr>
</tbody>
</table>

Having only about 33% of the CPU processing power in the test system compared to the average system is not an optimal solution. This is because it is likely that the CPU will be a determining factor in the performance test. I therefore decided to reduce the number of workstations to two on each hypervisor reducing the CPU difference to the average system further. I also hope that the hyper-threading of the hypervisors can make up for some of the difference.

<table>
<thead>
<tr>
<th></th>
<th>2 VM per hypervisor</th>
<th>Average system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>10 workstations</td>
<td>16 workstations</td>
</tr>
<tr>
<td>CPU</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>Memory</td>
<td>2 Giga bytes</td>
<td>2 Giga bytes</td>
</tr>
<tr>
<td>Network</td>
<td>500 Mbit</td>
<td>100 Mbit</td>
</tr>
</tbody>
</table>

Table 3 Average system comparison with 2 VM per hypervisor

Table 3 shows the environment that I will use for my performance test.

4.7 Identify Performance Acceptance Criteria

The acceptance criteria for the performance test is a bit different than having a customer setting the requirements. My acceptance criteria are to confirm hypothesis 2. When trying to confirm hypothesis 2 I will be limited in what I can change in the system. This is because it is a requirement that the adaptor works as a replacement for the DDL and uses an existing DDS solution, therefore it will not be possible to change the DDS code or redesign how the adaptor is used. I can only improve performance by changing the DDS configuration and changing the parameters the adaptor uses when calling functions in the DDS.

I will also be limited by time therefore I have set a fixed limit of 14 days to try and confirm the hypothesis.
### 4.8 Plan and Design Tests

To create a workload model, I started by looking at log files from existing installations. In the log file, I can see that simultaneously user interactions are uncommon in the DDL, normally there is just one user interaction at the time. Therefore, creating a workload model with simultaneously user interactions for the DDL would not be a typical scenario. Instead, I will perform a test on individual functions invoked by user interactions. This is the performance that a user will observe as they activate functions and wait for feedback on the HMI. The test should be as simple as possible so that it is easier to reproduce the results and make the results consistent while still being realistic.

In this chapter, I will define a typical usage scenario that stress the DDL and I will define how I can measure the performance of it.

#### 4.8.1 Typical usage scenario

The performance test should be a typical usage scenario that is representable for how the DDL is used and that still stress the DDL. To stress the DDL, I will look for a scenario where many topics are affected. The test will therefore be a spike test. To find this, I will investigate the usage of the adaptor.

The adaptor usage can be divided into two parts.

1. Setup topic publish and subscribe
2. Exchange topic samples

The first scenario is only performed after the system is started or when connections between workstations have been unavailable. I therefore do not consider this a typical scenario. I do think that it is very likely that the DDS will perform slower than the DDL in this scenario because of the peer-to-peer architecture of DDS. In the research papers “Bloom Filter Based Discovery Protocol for DDS Middleware” [DiscovProt] it is also indicated that DDS may have issues when the system gets larger. This research is described in the related work section.

Because the first scenario is not a typical scenario, I decided to perform a test on the exchanging of samples.

The exchanging of samples can be subdivided.

1. One publisher to one subscriber
2. One publisher to many subscribers

In Figure 19 the 2a scenario is shown where a single sample is send from a publisher to a subscriber in the DDL.
The same sequence for the DDS is shown in Figure 20.

From the above diagrams it is clear that the DDS is simpler than the DDL in this sequence and therefore the DDS is likely to be faster in this simple scenario. It is possible to create tests that will confirm this however I do not think this is interesting because it is likely to be a matter of a few milliseconds that one of the solutions will be faster than the other.

In the performance test I am looking for typical usage scenarios and in a real system the one to one scenario is also very uncommon. It is uncommon because a real system will have the same types of workstations as in the example with the HMI and IO workstations, but in a real system both IO and HMI workstations will be redundant, so there will always be at least two subscribers.

The one publisher to many subscribers scenario looks much like the 2a scenario except there will be more subscribers. This is shown in Figure 21.
The Figure 21 shows that the server sends asynchronously to the two subscribing workstations. It should be noted that the sever sends asynchronously the two workstations so they may be processing the sample simultaneously.
Because a scenario with multiple subscribers of each topic is typical in the DDL I will use this in the performance test.

**Test scenario**

Spike test with many topics and many subscribers of each topic.

The scenario can be created with different parameters that will affect the performance. Below I have listed the parameters that I want to consider for the performance test.

1. Distribution of subscribers in the system.
2. Server workstations in the system.
3. Subscribers on each workstation.
4. Size of each sample.
5. Number of samples to send.
6. Number of topics used to send samples.

I will go thru each of these and select parameters for the performance test.

**4.8.2 Distribution of subscribers in the system**

In the Identify Test Environment chapter I found that an average system has 16 workstations, however every workstation may not have subscribers for every topic. Some topics may just be subscribed by half of the workstations while others are subscribed by every workstation. Therefore, I need to decide how many subscribers I should have for each topic on each workstation. To understand how the subscribers are distributed in a real system I made a modified version of the DDLSERVER that counted how many samples every workstation sends to other workstations. I then activated a function that creates a spike and extracted the values counted.
Figure 22 shows a chord graph with samples received by each workstation in a spike scenario for a small system.

In the graph the name of the workstation is shown in the outer circle with the portion of the samples that it receives. The inside of the circle shows where the samples originated from. In the graph we can see that the server A is sending almost all the samples, the two IO workstations are sending a small portion of the samples. To simplify the test I will ignore this and just have one application sending samples.

**Test scenario**

Only one workstation sending samples

From the graph we also see that the receivers are fairly distributed in the entire system. The server A and B receives the highest number of samples so even though server A is sending the samples it is also subscribing them. For the test this means that the sending workstation should also subscribe to the topics.
Test scenario
Sending workstation must also subscribe

We also see that the two IO workstations receive the smallest number of samples. The HMI’s called HMI_1, HMI_2, LOGDB_1 and MW_1 all receive the same number of samples. All the workstations receive some samples therefore in my performance test every workstation should be subscribing to the topics that are updated.

Test scenario
Every workstation will have subscribers

In a larger system there will normally still only be two servers but the number of HMI and IO workstations will increase, this is shown in Figure 23

![Figure 23 Sample distribution in larger system](image)

In the figure we can see that the server workstations receive far less of the total samples. Therefore, in the performance test I will only look at the HMI and IO workstation subscriber distribution.

The HMI workstations receive about twice as many samples as the IO therefore I will create two types of subscribers in the performance test. One type that subscribes to twice as many topics as the other type.
4.8.3 Server workstations in the system.

In the chapter Typical usage scenario Figure 19 shows 3 workstations involved in sending a sample in the DDL. Figure 20 shows that only 2 workstations are needed in the DDS. Therefore I need to consider if the workstation that runs the DDLServer should be included when the DDS performance test is performed.

It would be reasonable to say that we want to performance test from the sample is published in the IO Application until it is displayed in the HMI application so therefore the DDS test should have fewer workstations. However, in the chord graph in Figure 22 we can see that the server workstations are very active in real systems, this is because the real systems will have applications on the server also. Real systems will look as shown in Figure 24 that also has an application on the Server workstation.

Therefore, in the spike test I will also have a subscribing application on the server workstations.

Test scenario
Server Workstations will have subscriber.
4.8.4 Subscribers on each workstation

The DDL uses the DDLClient to distribute samples on each workstation. It does this to reduce the samples send over the network. The performance scenario should reflect this so I have looked at how many local subscribers there are for all the topics on a workstation. Figure 25 shows how many topics have one local subscriber and how many have two. This is shown for both HMI and IO workstations.

![Subscriber distribution on workstations](image)

In the figure we can see that the for the IO workstation the DDLClient architecture is working as intended but for the HMI workstation this has no effect.

In the DDS it is possible that the that network utilization will be higher for topics with multiple local subscribers than it will be in the DDL. In the test I will therefore create two types of workstations one that has only a single local subscriber and one that has two local subscribers.

**Test scenario**

Two workstation types one with two local subscribers and one with one.

4.8.5 Size of each sample

The size of each sample will affect the performance because it affects the ratio between data and information about the data (meta data) for each sample send. When small samples are send the network capacity is likely to be waisted with sending meta data about each sample. Therefore, I want to select a size that is realistic in a real system. To select a size of each sample I analysed the data send from the DDLServer in the small system when the spike function is activated. I logged the total number of bytes send in samples and the total number of samples send. There were 123 kilobytes send in 4300 samples. This makes the average sample size about 28 bytes.

**Test scenario**

Sample size should be 28 bytes
4.8.6 Number of samples to send
To select the number of samples to send in the performance test. I have looked at the small system that was shown in Figure 22 and in the larger system that was shown in Figure 23. The samples send towards the server is 300 and 1500. It is obvious that the more samples that must be send the longer it will take. I want the performance test to stress the system so therefore I selected to send a bit more samples then the larger system had. By using a bit more samples I expect the difference between DDL and DDS to be highlighted. I decided to use 2000 samples.

Test scenario
Spike test with 2000 samples send.

4.8.7 Number of topics used to send samples
By inspecting logs, it is possible to see exactly what topics are updated when a spike function is activated. In the logs I found that some topics are updated twice and some only once. I therefore decided to send two samples for every topic to make sure I included this characteristic of the communication.

Test scenario
Spike test with 1000 topics updated.

4.8.8 How to measure response time
The applications will run on different workstations therefore it is difficult to precisely measure the time it takes to send all the samples, because the clocks would need to be synchronized very precisely. Instead I decided to use the sending application to measure the time by making the subscribing applications send a reply once they have received all the samples. The time measured will be longer than the actual sending time, because the reply will take time also, however this will be the same for both the DDL and the DDS and therefore this should not affect the results significantly.

Test scenario
Subscribers will update a reply topic once they have received all updates.

4.8.9 Test scenario
In this chapter I will describe how I will put all parameters together for the performance test. The performance test will be executed in the environment shown in Figure 26.
The figure shows an artefact called DDL/DDS in every application, this is the part that will be exchanged between the two performance tests. The figure does not show the connections used by the DDL or DDS because this is different for each of them.

The figure has one Publish application this is the application that will initiate the spike test and measuring the response times.

The subscribe applications will subscribe to many topics and publish a single sample once they have received a sample on all the topics. This is shown in Figure 27.
The publish application will simply measure the time from it starts sending out samples until the reply sample is received. Because there are 15 Applications there is 15 time measurements for each test run.

The subscribers will use different numbers of topics. Table 4 shows how each is configured.

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Workstation</th>
<th>Application</th>
<th>Subscribe topic count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>Publish application</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>10</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>11</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>13</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>14</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4 Test configuration*
4.9 Configure Test Environment
This activity is about setting up the tools used when the test is executed. This includes the tasks listed below.

1. Acquiring workstations and switches
2. Setting up the network hardware
3. Coordinating a range of IP addresses
4. Handle version compatibility between monitoring software and operating system.
5. Prepare test environment

Because I have decided to use an existing test system point 1 to 3 is already handled.
For point 4 I will not be installing software to monitor the resources on the virtual machines because I can just use the monitoring tools already provided in the hypervisor.
To prepare the test environment for my test I decided to create batch files that I can use to easily copy the artefacts used for testing to each workstation. The artefacts include the DDS adaptor or DDL version, a test application and other required DLLs. Using a batch file helps avoid manual mistakes like forgetting to copy a new version to one of the workstations. When starting the test, I will still need to go to each VM and execute the test application but this will not take long and is less error prone.
To ensure that I get consistent results I decided that for each of the hypervisors I will shut down all other virtual machines when I execute the test.

4.10 Implement Test Design
For my test there is no reason to complicate the test by going thru a GUI instead the test scenario can be implemented directly towards the DDL API. I will do this by writing C++ code that execute the test scenario and displays the result.
To quickly get started I will use an existing application and create a small GUI to execute the test and show the result. The test GUI is shown in Figure 28
The test is made so that the subscribers only reply if they have received all the samples they expect. This ensures that it is easy to see if some samples are lost.

4.11 Execute Tests
When running the performance test, I will
1. Reset the environment by restarting the test applications. This will ensure that previous test runs will not affect the result.
2. Then I will ensure that memory caching is warmed up by running the test once without using the result.
3. Then I will execute the test 5 times to ensure I get consistent results.
4. After the test is executed I will look in log files for unexpected errors.

4.12 Analyze, Report and Retest
To compare the performance of the adaptor I first need to establish the performance of the DDL. The results of running the performance test on the DDL is shown in Figure 29.
The figure shows the time for each of the 15 applications in each of the 5 test runs. In the figure we can see that there is some variation in the results for each test. The largest variation is for application 10 where the lowest time is 343 milliseconds in test 3 and the highest time is 655 milliseconds in test 4. This is not very consistent results as I would have liked to see. However, for every test run there is at least one application that shows a time around 600 milliseconds. Therefore, I believe the variation is because the server is limiting the performance and it is random which application will finish last.

4.12.1 First iteration
The performance test result from the DDL and the first version of the DDS adaptor is shown in Figure 30.

The figure shows the minimum, average and maximum of the 15 times produced for each test run. For the DDS the maximum time measured is very inconsistent. When testing it is not good to have inconsistent results because we want to be able to reproduce and compare the results. I therefore wanted to understand why the performance of some applications in some tests was above 10 seconds while in others all applications were below 2 seconds.
The first step was to look at the details of the results. These are shown in Figure 31.

![Figure 31 Individual DDS times in first version](image)

The figure shows that it is application 1 that has very bad performance sometimes. Running additional tests also confirmed this. I found this very strange because application 1 is the publishing application and I would therefore expect it to have the fastest performance because it would not need to send the samples over the network. Because the performance difference is 8 seconds I think it can be considered a defect. To find defects we can use the Scientific debugging method described by Andreas Zeller [Zeller]. This method describes an iteration cycle that includes the activities listed below.

1. Create or refine hypothesis
2. Make prediction
3. Create experiment
4. Observation / Conclusion

First, I create a hypothesis about the defect, then I make a prediction based on the hypothesis and create an experiment that will show if the prediction is right. Finally, I look at the result and if the hypothesis is right I refine it and if it is not I create a new hypothesis.

Because I do not know for sure that the long delay is a defect I will refrain from using the term defect and instead use the term delay.

Zeller also describes seven steps for debugging in general, the list can help categorize the activities that should be done when trying to find a defect. Activities are listed below.

1. Track the problem
2. Reproduce the problem
3. Automate and simplify the test case
4. Find possible infection origins
5. Focus on the most likely origins
6. Isolate the origin of the infection
7. Correct the defect

The first step is to register the problem, and this report will serve for that. If I do find that the problem is related to RTI’s DDS solution I will report it to them also.
The next step is to reproduce the problem. I found that I can do this by running the performance test a few times. The only action I do when running the test is to push the start button. However, because the delay only happens sometimes there must be a random element involved that is outside my control. The random element could be thread scheduling, network activity or something else. Because I cannot reproduce the delay consistently I will have to look at the likelihood when trying to reproduce the problem.

The third step is to automate producing the problem. Because I must work with likelihood I decided to extend my test setup with an automatic mode that would run the test continuously with a small delay between each run. This will allow me to get a much better statically perspective of how likely it is that the delay will happen. It will also eliminate the random time element between each test run that might affect the results.

After running the tests for a longer period I found that the likelihood of the delay was 51%. I will need this to evaluate the hypothesis I create.

When I do the four Scientific debugging activities the steps 4 and 5 is a part of this process. When I formulate a hypothesis, I will be considering the possible infection origins and focus on the most likely origins. The Scientific debugging iterations will end once I have isolated the origin of the infection.

In the next chapters I will describe each iteration in the debugging process.

4.12.1.1 Delay hypothesis 1
Because the delay is in the publishing application I will start by trying to confirm that the delay is related only to internal communication in that application. I do not think it is logical that internal communication should be delayed for up to 8 seconds because of external communication. Confirming this will also simplify recreating the delay making the next iterations easier to do.

<table>
<thead>
<tr>
<th>Delay hypothesis</th>
<th>Delay is related to sending a sample internally in the publishing application and the number of other applications will not affect the likelihood of the delay.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>The delay statistics is the same when there are fewer applications.</td>
</tr>
<tr>
<td>Experiment</td>
<td>Test with the first three applications in the test setup.</td>
</tr>
<tr>
<td>Observation</td>
<td>147 times out of 2639 times = 5%</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis rejected. The delay must be related to the number of applications.</td>
</tr>
</tbody>
</table>

4.12.1.2 Delay hypothesis 2
The delay can be created because of a defect in any part of the test. The test consists of four parts.

1. Performance test code
2. Adaptor code
The delay can be a result of a defect in the way the performance test is created. For instance, it could be because of something in how the reply topic is implemented. Because the delay is occurring in the publishing application it is possible to simplify the test by removing the reply topic from the test and using a memory variable instead. It is also possible that the delay is because of a defect in the adaptor implementation. For instance, a thread being locked and preventing the DDS from notifying the adaptor. Therefore, I made a hypothesis that it was a defect in the adaptor code or the test code. To confirm this, I needed to create an experiment where the code that used the DDS was as simple as possible. For this I decided to use the RTI HelloWorld example and extended it to look like my performance test. The topic count, sample count and QoS settings should be as close to the original performance test as possible to ensure that if the defect is within the DDS code or QoS settings then it will still be there.

<table>
<thead>
<tr>
<th>Delay hypothesis</th>
<th>Delay is caused by some incorrect usage of the DDS API or in the way the results are measured in the test code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>HelloWorld example will not show the same behavior with the same number of participants, topics and update pattern.</td>
</tr>
<tr>
<td>Experiment</td>
<td>Extend HelloWorld example to 1000 topics and two updates. Test with 10 participants and the same QoS settings.</td>
</tr>
<tr>
<td>Observation</td>
<td>5 times out of 20 times = 25%</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis rejected. Delay must be caused by DDS defect or wrong QoS settings.</td>
</tr>
</tbody>
</table>

4.12.1.3 Delay hypothesis 3
Having eliminated my test code and the adaptor code. The delay would have to be because of either a defect in the DDS or a wrong or missing QoS setting. Because I do not have access to the DDS code it is not possible to dig deeper into this, I therefore must continue by looking at the QoS settings.

The QoS settings contains only a setting for reliability and one for history. These setting describe the behaviour that I want the DDS topics to have and therefore the defect cannot be with these settings. It is however possible that I am missing a setting that could prevent the delay but before I can make any hypothesis about a missing setting I would need a better understanding of what is happening when I execute the test.

In the first delay hypothesis I observed that the likelihood of the delay would decrease when the number of applications decreased. The only way the other applications can affect the publishing application is with the network communication that they are sending. I therefore wanted to make a hypothesis that would change some QoS setting related to the network communication.
To do this I needed to understand what was happening at the network level. For this I decided to use Wireshark [wireshark] to analyse the network traffic. Wireshark is a program that allows you to see the ethernet packets that are send on a network. Instead of using Wireshark on the performance test I decided to create a minimum performance test for two reasons.

1. If I install Wireshark on the Workstations used in the performance test I am very likely to be affecting the test result because of the observer effect.
2. With 15 participants and thousands of samples being exchanged there will be many packets and this will make it difficult to see what is happening for a single topic.

I therefore decided to create a minimum performance test with only two workstations and two topics. The number of samples on each topic would still be two and the subscriber would still use a reply topic just like the original performance test. I decided to use two topics and not one to ensure that if there were any optimizing being done because of multiple topics these would still be seen.

In Figure 32 all the packets send in this minimum performance test is shown.

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000</td>
<td>192.168.1.122</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>142</td>
<td>INFO_TS, DATA</td>
</tr>
<tr>
<td>2</td>
<td>0.00001</td>
<td>192.168.1.122</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>142</td>
<td>INFO_TS, DATA</td>
</tr>
<tr>
<td>3</td>
<td>0.00002</td>
<td>192.168.1.122</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>142</td>
<td>INFO_TS, DATA</td>
</tr>
<tr>
<td>4</td>
<td>0.00003</td>
<td>192.168.1.122</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>142</td>
<td>INFO_TS, DATA</td>
</tr>
<tr>
<td>5</td>
<td>0.00004</td>
<td>192.168.1.157</td>
<td>192.168.1.122</td>
<td>RTPS</td>
<td>159</td>
<td>INFO_TS, DATA</td>
</tr>
<tr>
<td>6</td>
<td>0.03525</td>
<td>192.168.1.157</td>
<td>192.168.1.122</td>
<td>RTPS</td>
<td>110</td>
<td>INFO_DST, HEARTBEAT</td>
</tr>
<tr>
<td>7</td>
<td>0.03761</td>
<td>192.168.1.157</td>
<td>192.168.1.122</td>
<td>RTPS</td>
<td>100</td>
<td>INFO_DST, ACKNACK</td>
</tr>
<tr>
<td>8</td>
<td>0.16212</td>
<td>192.168.1.122</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>110</td>
<td>INFO_DST, HEARTBEAT</td>
</tr>
<tr>
<td>9</td>
<td>0.16233</td>
<td>192.168.1.157</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>100</td>
<td>INFO_DST, ACKNACK</td>
</tr>
<tr>
<td>10</td>
<td>0.16246</td>
<td>192.168.1.157</td>
<td>192.168.1.122</td>
<td>RTPS</td>
<td>110</td>
<td>INFO_DST, ACKNACK</td>
</tr>
<tr>
<td>11</td>
<td>0.16254</td>
<td>192.168.1.157</td>
<td>192.168.1.157</td>
<td>RTPS</td>
<td>110</td>
<td>INFO_DST, ACKNACK</td>
</tr>
</tbody>
</table>

*Figure 32 Wireshark screenshot*

By looking of the packet contents and the source and destination I could see that the first 4 DATA packets are the publisher updating two topics with two samples. In the screenshot these are just called DATA but I will use the term sample packets for these to avoid confusion.

Packet 5 is the subscriber updating the reply topic with a sample packet. These 5 sample packets are to be expected based on the sequence shown in Figure 27, but after this there are three heartbeat packets and three acknack packets these are not so obvious. After searching the RTI website I found a description of these packets [RtiReliableProt]. They are needed because the protocol is using unreliable UDP and as described in Adaptor requirement 1 the topics are configured to be reliable so the publisher needs to ensure that packets are received by the subscriber. The heartbeat packet is a request from the publisher to the subscriber to acknowledge that it has received the latest sample on a topic and the acknack packet is the reply indicating it is received. There are three of each because there are three topics in use. The heartbeat packet will request acknowledgement of both samples send.

From this we see that for every topic there is 4 packets send for each subscriber.

With this knowledge it was possible to create a hypothesis that the delay was because of the processing of the heartbeat and acknack packets in the publishing application. Because there is
a small delay from when the sample packet is send until the heartbeat is send it must involve some timer or thread and that might be causing a race condition or other types of threading problems. When the application is processing the incoming heartbeats while sending samples this may also create threading problems. Therefore I wanted to try and simplify the threading within the publishing application.

After searching RTI’s website I found that the behaviour of the heartbeat packet can be controlled from the configuration file with the max_send_window_size and heartbeats_per_max_samples keywords [RtiControlHb]. By setting both these parameters to the same values the heartbeat would be send with every sample packet instead of as a separate packet. This would therefore simplify the sending of the heartbeat. I select to use 10 samples as the send window size, however the actual number should not be important for my test.

I also found that the time from when an acknack packet is requested until the subscriber responds can be configured with the min_heartbeat_response_delay and max_heartbeat_response_delay. The subscriber will select a random time between these two values for sending the acknack packet. By selecting a min_heartbeat_response_delay that is higher than the time it takes the publisher to send all the samples I can ensure that sending within the publisher is not affected by incoming acknack packets.

By setting these two QoS settings the only processing that the publishing application will have to do while sending is the handling of reply topic samples from the other applications.

I then added these parameters to the configuration by creating a new profile that inherited from my first version and changing the parameters in the new profile. The new profile is shown in DDS Profile 3.

```xml
<qos_profile name="DDLQoS_Simple" base_name="DDLQoS">  
  <protocol>  
    <rtps_reliable_writer>  
      <max_send_window_size>10</max_send_window_size>  
      <min_send_window_size>10</min_send_window_size>  
      <heartbeats_per_max_samples>10</heartbeats_per_max_samples>  
    </rtps_reliable_writer>  
  </protocol>  
  <datareader_qos>  
    <protocol>  
      <rtps_reliable_reader>  
        <min_heartbeat_response_delay>  
          <sec>0</sec>  
          <nanosec>200000000</nanosec>  
        </min_heartbeat_response_delay>  
        <max_heartbeat_response_delay>  
          <sec>0</sec>  
          <nanosec>400000000</nanosec>  
        </max_heartbeat_response_delay>  
      </rtps_reliable_reader>  
    </protocol>  
  </datareader_qos>  
</qos_profile>
```

**DDS Profile 3 DDLQoS_Simple**
I decided to send the acknack packets between 2 and 4 seconds after the sample and heartbeat packet. This is a high delay but for the performance test it ensures that processing in the publishing application is spread out over time. In a real system it may not be desirable with such high values because it will mean that if packets are lost it will take a longer time until the packet is retransmitted.

| Delay hypothesis | Delay is related to the processing of heartbeat and acknack packets while sending other samples. |
| Prediction        | Delay will not happen when processing of heartbeat is done with sample sending and when acknack packet is delays until the sending is finished. |
| Experiment        | Change QoS settings for network packets |
| Observation       | 15 times out of 1435 times = 1% |
| Conclusion        | Hypothesis accepted |

The observation does not align completely with the prediction. But the change clearly made a big difference in the likelihood of the delay. Despite the delay not being eliminated completely I still think the hypothesis may be valid, there may just be other elements that is also affecting the result that I did not find.

Without access to the source code of the DDS it will not be possible to track the infection back to a defect. I therefore decided that I would have to accept that in 1% of my tests the response time would be extra high.

I reported the suspected defect on the RTI community [RtiForumSlow] but at the time of writing they did not confirm that this was in fact a defect.

4.12.2 Second iteration
The results after running the performance test with DDS Profile 3 DDLQoS_Simple is shown in Figure 33

Figure 33 DDL and DDS second version comparison
In the figure we can see that the maximum time measured for the 5 tests, are now much more consistent. By looking at the individual times from the 15 applications, which is shown in Figure 34, it is possible to see that the variation for each application in the 5 test runs is very small. The largest variation for a single application is 80 milliseconds, which I considered an acceptable value for a non real-time system, as Windows based PC's using IP based communication.

![Graph showing DDS times in second version](image)

**Figure 34 Individual DDS times in second version**

In Figure 33 we can see that the DDS adaptor is slower than the DDL. Therefore, I need to figure out if I can improve this to confirm Hypothesis 2. The analysis of the communication in the first iteration showed that for each topic there was first 4 packets send for each subscriber. This was then reduced to 3 by sending the heartbeat packet with every sample packet. From this we can calculate the number of packets that would be sent in the performance test.

The performance test has 10 applications with 1000 topics and 5 applications with 500 topics, therefore we can calculate that there will be at least \(((10 \times 1000) + (5 \times 500)) \times 3 = 37500\) ethernet packets send. There may be more packets in case some packets are dropped.

For the DDL this is different because it uses TCP/IP. TCP/IP connections use the Nagle's algorithm [RFC896], this algorithm prevents many small packets on the network by delaying some packets and combining these into larger packets. When this is done the utilization of the physical network is much higher.

After searching the RTI website I found an article about QoS settings [RtiQoSParms]. This article describes a Batch QoS setting that will configure the DDS to collect multiple user data samples to be sent in a single network packet. Therefore, using this should give the same effect as the Nagle's algorithm.

For the adaptor to use this a batch section must be added to the configuration. This section contains a `max_flush_delay` keyword that controls the delay the DDS will use to collect multiple samples into a single network packet. This configuration is shown in DDS Profile 4.
I decided to use a delay of 20 milliseconds as flush delay because I wanted a value that would not affect the performance too much.

4.12.3 Third iteration
The results after running the performance test with DDS Profile 4 DDLQoS_Batch is shown in Figure 35.

In the figure we see that the performance of the adaptor is getting close to the DDL performance. We can also see that the maximum time measured is more consistent and by looking at the individual times from the 15 applications I can also see that the largest variation was reduced to only 30 milliseconds.

The average performance is still not quite as fast as the DDL. Therefore, I still need to improve the performance to confirm hypothesis 2.

Before trying to optimize even more I wanted to confirm that the batch configuration and packets rescheduling that I did in the two first iterations were working as I thought they would. To do this I again used Wireshark with the same minimum performance test.
In Figure 36 we can see that both samples and the heartbeat is being send together in one packet and that the acknack is more than 2 seconds after the heartbeat is send. These were the changes in the first iteration and they work as intended. There are however 2 batch packets from the publisher where I would have expected to only see one big packet with all 4 samples. After looking at the packets it was clear that the batch command only allowed samples on the same topic to be batched together. Compared to the Nagle's algorithm where samples on all topics between the two applications are bundled together this is not as efficient.

For the performance test this means that there are still ((10 * 1000) + (5 * 500)) * 2 = 25000 ethernet packets send.

To improve the performance further I considered two options.

1. Use TCP/IP to get the advantage of Nagle's algorithm.
2. Use multicast to reduce number of packets send.

I decided to pursue the multicast option for two reasons.

1. When I selected the test system I used fewer workstations than the average system had. When the number of workstations increase the number of sample packets required to send the samples will be constant when using multicast. Therefore, I expect that the DDS response time should also be constant when the number of workstations increase unlike the DDL response time that would increase. The acknack packets will still mean that the packets send will increase when the number of workstations increase, however this will not affect the response time in a test like mine because acknack packets are delayed until the test is finished. Therefore I expect the scalability to be better with multicast.
2. Because the DDL also use TCP/IP the results are likely to only be similar or maybe slightly slower. Therefore I am unlikely to clearly confirm the hypothesis.

To configure the adaptor to use multicast the <multicast> sections must be added. This is shown in DDS Profile 5.

```xml
<qos_profile name="DDLQoS_Batch_Multicast" base_name="DDLQoS_Batch">
  <datareader_qos>
    <multicast>
      <value>
        <element>
          <receive_address>239.255.5.1</receive_address>
        </element>
      </value>
    </multicast>
  </datareader_qos>
</qos_profile>
```

**DDS Profile 5 DDLQoS_Batch_Multicast**
I selected the multicast address 239.255.5.1 because it was not used on the test network, also the DDS adaptor will use a port number to ensure that even if another program is using the same multicast address it will not be a problem.

### 4.12.4 Forth iteration

The results after running the performance test with DDS Profile 5 DDLQoS_Batch_Multicast is show in Figure 37.

![Graph showing performance comparison between DDL and DDS for forth iteration](image)

**Figure 37 DDL and DDS forth version comparison**

In the figure it is now clear that the adaptor is faster than the DDL in minimum, average and maximum. The average time for the DDL was 426 milliseconds and the adaptor average was 287 milliseconds therefore it is 32% faster.

To make sure the results were correct I looked at the individual times for each application. This is shown in Figure 38.
In the figure all applications that only subscribe 500 topics have a lower time and this gives me confidence that the test is executed correctly.

The figure shows consistent results across virtual machines and hypervisors, this is nice because it shows that it is the network or the publishing applications ability to send samples that is limiting the results.

The largest variation was reduced to only 15 milliseconds. This indicates that the multicast is working as intended. Compared to the DDL where the variation was 300 milliseconds this is also a significant improvement.
5 Related work

There is no related work that compares the DDL to DDS. But in my research, I found articles about DDS performance. Below I will describe these articles and explain why they are interesting in relation to the work that I did.

5.1 Evaluating the Performance of DDS

"Evaluating the Performance of Publish/Subscribe Platforms for Information Management in Distributed Real-time and Embedded Systems" [EvalPerDDS] is a university paper on the performance of DDS. The paper compares three different DDS architectures and other messaging middleware solutions. The other solutions that is compared is Java Message Service (JMS), Simple Object Access Protocol (SOAP) and CORBA Notification Service. The paper concludes that the DDS performs better than the other solutions for three reasons.

1. DDS’s communication model provides a range of QoS parameters that allow applications to control many aspects of data delivery in a network.
2. Implementations can be optimized heavily.
3. DDS can be configured to leverage fast transports, e.g., using shared memory to minimize data copies within a single node.

The DDL uses a client-server architecture just like JMS therefore these are comparable. For my work this is interesting because the conclusion that DDS is faster than a client-server architecture aligns with my conclusion.

The paper is comparing three different DDS architectures but it does not write what DDS providers were used for the three architectures. Therefore, I cannot confirm if any the tested DDS providers is RTI. I can confirm that the architecture of the RTI solution has the same architecture as the DDS with the best performance in the tests they performed. Therefore, it is likely that the RTI solution that I have used is one of the better DDS providers.

5.2 Discovery Protocol for DDS

"Bloom Filter Based Discovery Protocol for DDS Middleware" [DiscovProt] is a university paper that addresses the design and evaluation of a Simple Discovery Protocol alternative called SDPBloom. The Simple Discovery Protocol (SDP) is the protocol used to setup the connection between publishers and subscribers. They propose the SDPBloom alternative to overcome the scalability limitations found in SDP. They conclude that.

- SDPBloom can improve the discovery process in DDS applications (in terms of network load and node resource consumption)

For my work this is interesting because I did not investigate the setup phase of a DDS solution. This work therefore highlights that it will also be necessary to investigate this phase.
6 Conclusion

6.1 Hypothesis 1
In this report I have investigated the hypothesis below.

It is possible to create a DDS adaptor with less than 1000 lines of code for the DDL that will have publish, subscribe and update functionality.

To investigate the hypothesis, I did an architectural reconstruction of the DDL to understand the publish, subscribe and update functionally. Then I evaluated different DDS providers, based on license and what version of the DCPS, RTPS and XTypes specification they complied to, to find a provider that was suitable to test the hypothesis. After selecting a DDS provider, I created UML sequence diagrams from examples and online documentation to compare to the DDL. This allowed me to understand the difference in publish, subscribe and update functionally. With a clear understanding of the difference between DDL and DDS I created an adaptor between the two messaging systems.

I did not want to create a new DDL on top of another messaging system therefore I set a limit of 1000 lines to code to ensure that the adaptor was simple. To confirm this, I have simply counted the number of source code lines in my adaptor in the publish, subscribe and update functionality. This excludes source code for functions that are needed for DDL API combability. This count was 678 lines and the hypothesis is therefore confirmed.

When looking at the sequence diagrams for the DDL and DDS they have many similarities. I think that that underlying intension of creating a simple adaptor is also achieved when looking at the relative simple implementation in the adaptor. It was only the requirement to find the topic type from the topic name described in 4.4.4 that was a bit more complicated.

6.2 Hypothesis 2
In this report I have investigated the hypothesis below.

A DDS solution via an adaptor pattern can provide the same or better topic update performance in a scenario that will stress the DDL.

To investigate the hypothesis, I used Microsoft’s list of performance testing activities to structure the work.

First, I analysed existing DDL based systems to find a good average system that was representable for the different systems where the DDL was used. Then I compared the average system to the system I could use for testing to understand and minimize the difference. Then I analysed a scenario that stress the DDL to create a simple realistic scenario that I could use in the performance test. Then I implemented the performance test in C++ code and iteratively optimized the performance of the adaptor by changing QoS settings.
Finally, I confirmed the hypothesis showing that the DDS from RTI combined with my adaptor implementation was 32% faster than the propitiatory DDL solution. The main reason for the better performance was usage of multicast packets. Multicast is a feature of most routers today however it should be noted that this may not be available in all the networks where the DDL is operating. In these networks there are other options like using TCP/IP or unicast UDP/IP that had only slightly slower average response times compared to the DDL. Confirming the hypothesis by using multicast means that for systems that are bigger than the test system the results are likely to be better, because multicast will scale better than unicast. For systems that are smaller it is likely that the use of multicast will not improve performance as much. For these systems however, performance is typically not an issue because they are smaller and have fewer topics.

This project has shown that the DDS is a potential candidate for exchanging the DDL. It has shown that by optimizing the QoS settings of the DDS network communication the performance can be improved significantly. The DDS and adaptor solution also showed more consistent test results. This is a nice bonus because it will mean that the systems will have a higher determinism.
List of figures

Figure 1 DDL Architecture ................................................................. 3
Figure 2 DDS Architecture .................................................................. 5
Figure 3 DDL Connecting .................................................................... 11
Figure 4 DDL Publish ........................................................................... 12
Figure 5 DDL Send update ................................................................... 13
Figure 6 DDL Subscribe ....................................................................... 14
Figure 7 DDL Receive update ............................................................... 14
Figure 8 DDL API Classes ................................................................. 15
Figure 9 DDS Connecting ................................................................. 19
Figure 10 DDS Register topic type ...................................................... 21
Figure 11 DDS Publish ...................................................................... 22
Figure 12 DDS Send update ............................................................... 23
Figure 13 DDS Subscribe ................................................................... 24
Figure 14 DDS Receive update .......................................................... 25
Figure 15 Object model with one topic to one type name ...................... 28
Figure 16 Object model with type name from members ........................ 29
Figure 17 Publish before subscribe .................................................... 30
Figure 18 Subscribe before publish .................................................... 31
Figure 19 Send sample to one subscriber in DDL ............................... 35
Figure 20 Send sample to one subscriber in DDS ............................... 35
Figure 21 Send sample to two subscribers in DDL .............................. 36
Figure 22 Sample distribution in small system ..................................... 37
Figure 23 Sample distribution in larger system ................................... 38
Figure 24 Real system deployment ..................................................... 39
Figure 25 Subscriber distribution on workstations .............................. 40
Figure 26 Test environment ............................................................... 42
Figure 27 Test sequence .................................................................... 43
Figure 28 Performance test GUI ........................................................... 45
Figure 29 Individual DDL times .......................................................... 46
Figure 30 DDL and DDS first version comparison ............................... 46
Figure 31 Individual DDS times in first version ................................. 47
Figure 32 Wireshark screenshot .......................................................... 50
Figure 33 DDL and DDS second version comparison ......................... 52
Figure 34 Individual DDS times in second version ............................. 53
Figure 35 DDL and DDS third version comparison ............................. 54
Figure 36 Wireshark analysis with batch ............................................ 55
Figure 37 DDL and DDS forth version comparison ............................. 56
Figure 38 Individual DDS times in forth version ................................. 57
8 References

[Christensen]
Flexible, Reliable Software: Design Patterns and Frameworks, 2009 Henrik Bærbak Christensen

[ConnextDDS]
https://www.rti.com/products/dds

[ConnextDDSXtype]

[CoreDXDDS]
http://www.twinoakscomputing.com/coredx

[DDS]
OMG DDS main page
http://portals.omg.org/dds/

[DDSCore]
OMG DDS Standards
http://portals.omg.org/dds/omg-dds-standard/

[DDSProt]
https://www.omg.org/spec/DDSI-RTPS/2.2/PDF

[DDSSpec]
OMG, Data Distribution Service for Real-time Systems specification, version 1.4, 2015
https://www.omg.org/spec/DDS/1.4/PDF

[DiscovProt]
https://www.researchgate.net/publication/220379098_Bloom_filter-based_discovery_protocol_for_DDS_middleware

[eProsimax]

[eProsimax_Type]

[EvalPerDDS]
Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas C Schmidt.
Evaluating the Performance of Publish/Subcribe Platforms for Information Management in Distributed Real-time and Embedded Systems.
http://www.dre.vanderbilt.edu/~schmidt/PDF/RTAS_DDS.pdf
[Symphony]

[UML]
OMG, Unified Modeling Language, Version 2.5.1
https://www.omg.org/spec/UML/2.5.1/PDF

[wireshark]
https://www.wireshark.org/

[Zeller]
Why Programs Fail (Second Edition), A Guide to Systematic Debugging, Andreas Zeller