MASTER'S THESIS

A framework for supporting distributed video content applications

by
Koen S.W. van Langen
Mathijs Opdam

Supervisors:
Prof. dr. Johan J. Lukkien
Dr. Rudolf H. Mak
Dr. Bojan Orlic

April 22, 2010
Abstract

In general video content applications are computationally expensive. It might be that a single hardware platform does not provide sufficient resources needed by video content applications. Distribution of such applications over multiple hardware platforms enables assigning additional resources.

Currently ViNotion develops monolithic video content applications. The disadvantage of monolithic software development is that it lacks flexibility, especially when distribution is required. We investigate how monolithic video applications can be broken down into units that can be distributed across a network and allow transparent communication without degradation of functionalities. In addition we propose facilities for centralized remote management. In order to ensure a uniform approach of developing and distributing video content applications we propose a framework.

By selecting a representative monolithic video content application, we investigate actions that are necessary for distribution. These actions are grouped into different scenarios and allow identification of the requirements.

We conduct a survey to investigate available frameworks that could serve as a basis for our desired framework. During this survey the available frameworks will be examined with respect to the identified requirements. Once a suitable framework has been selected, it needs to be analyzed in more detail to identify discrepancies between the offered functionalities and the requirements imposed on the desired framework. For missing functionalities we propose solutions and introduce new concepts.

The result is a dedicated framework for distributing video content applications. The framework introduces an abstraction layer to hide the underlying middleware, allowing a systems engineer to focus on designing distributed video content applications.
Preface

This Master project is performed by two students of Eindhoven University of Technology enlisted with the Department of Mathematics and Computer Science following the master program in Embedded Systems. This thesis is the result of our graduation period at ViNotion B.V.. The Master project is executed within a time span of 7 and a half months over the period of 31 August 2009 through 21 April 2010.

With this thesis we try to create an overview of our activities performed during our Master project. This thesis is initially meant for our school supervisors, Prof. dr. Johan J. Lukkien and Dr. Rudolf H. Mak. It is also meant for employees of ViNotion as a reference. In this thesis we assume the reader to have knowledge of Computer Science. Some definitions on Computer Science are therefore assumed to be known.

Hereby we want to thank the employees of ViNotion and Ir. Rick J.J. Peerlings and Dr. ir. Egbert G.T. Jaspers in particular for their pleasant cooperation and for creating a pleasant working atmosphere. Special thanks goes to Dr. Rudolf H. Mak, Dr. Bojan Orlic and Prof. dr. Johan J. Lukkien for helping us supervising and completing the assignment on behalf of the System Architecture and Networking group.

Finally we want to thank our girlfriends, Anouk and Jessica, for their continuous support and patience during our Master period and our parents for supporting us and making it possible for us to follow and complete our Master Embedded Systems.


Koen van Langen and Mathijs Opdam
# Contents

**Preface** iii

**List of Figures** x

**List of Tables** xi

**Acronyms** xiii

1 Introduction 1
   1.1 Context and background ............................... 1
   1.2 Problem description .................................. 1
   1.3 Project goals ........................................ 1
   1.4 Approach ............................................. 2
   1.5 Outline .............................................. 2

2 Requirements of the *ViFramework* 5
   2.1 Leading example ....................................... 5
   2.2 *ViFramework* concepts and entities ................. 6
      2.2.1 Software component ............................... 6
      2.2.2 Docks ............................................. 7
      2.2.3 Dock manager ...................................... 8
      2.2.4 Hosts ............................................. 8
      2.2.5 Dock repository ................................... 8
      2.2.6 Distribution manager .............................. 8
      2.2.7 Application manager ............................... 8
      2.2.8 Webserver ......................................... 8
      2.2.9 System engineer ................................... 8
   2.3 Short overview of architectural scenarios .......... 8
   2.4 Architectural scenarios ............................... 9
      2.4.1 Scenario 1: Decomposition of the application into components ... 9
      2.4.2 Scenario 2: Aggregate components of an application into docks ... 10
      2.4.3 Scenario 3: Initial mapping of docks on hosts ................ 11
      2.4.4 Scenario 4: Host preparation ........................ 12
      2.4.5 Scenario 5: Deploying docks ........................ 13
      2.4.6 Scenario 6: Real-time dock performance extraction ............. 14
      2.4.7 Scenario 7: Manage connections between docks ................ 15
      2.4.8 Scenario 8: Initiate a redistribution of the application ....... 17
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Person entering the designated restricted area from different sides.</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>The complete virtual fencing system.</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>The architecture of the ViFramework. Note that we distinguish between an</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>application manager, for managing application parameters and a distribution</td>
<td></td>
</tr>
<tr>
<td></td>
<td>manager, for managing the distribution of the application. This is to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>emphasize the three different kinds of information flow: 1. application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data (black/solid arrows), 2. application control data (red/dashed arrows),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. distribution control data (blue/dotted arrows).</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Virtual fencing application: Communicated data.</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Decomposition of the VCA component into two subcomponents.</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Mapping the components into Docks.</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>Distribution of docks over several hosts in the network.</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>Distribution of the Managers several hosts in the network.</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>Add a dock to a host using the distribution manager, the dock repository and</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>the dock manager.</td>
<td></td>
</tr>
<tr>
<td>2.10</td>
<td>Retrieve performance metrics from the Dock managers.</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>Maintains (data)connections between docks with the use of the distribution</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>manager.</td>
<td></td>
</tr>
<tr>
<td>2.12</td>
<td>Remotely configure the application on site through a webserver.</td>
<td>18</td>
</tr>
<tr>
<td>2.13</td>
<td>Remotely get/set the parameters of the application by connecting with a web-</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>server which communicates with the application manager.</td>
<td></td>
</tr>
<tr>
<td>2.14</td>
<td>Failure scenario: data connection time out, which indicates a broken</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>connection or the unavailability of a dock.</td>
<td></td>
</tr>
<tr>
<td>2.15</td>
<td>Failure scenario: host is unavailable.</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>A layer view of an application running on top of middleware.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>DDS components, taken from Farabaugh et al [17]</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Architecture of capturing and displaying applications.</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Single stream 320x240pixels (24 bits).</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>Up to 4 streams of 320x240 pixels.</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Global architecture of the libViStreaming library, taken from the libViSte-</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>mming documentation.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Layering of the libViStreaming library.</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Grouping software components into docks.</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>A cyclic application topology.</td>
<td>46</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fault model showing connectivity, availability and maintainability errors.</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Pragmatic requirements priorities</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>Composition requirements priorities</td>
<td>27</td>
</tr>
<tr>
<td>2.4</td>
<td>Deployment and distribution requirements priorities</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Survey, framework properties</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Platform specifications</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Tool and library versions</td>
<td>34</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview of the scenarios and their support by the ViFramework</td>
<td>70</td>
</tr>
<tr>
<td>8.2</td>
<td>Responsibility Matrix</td>
<td>73</td>
</tr>
</tbody>
</table>
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Definition Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CANTATA</td>
<td>Content Aware Networked systems Toward Advanced and Tailored Assistance</td>
</tr>
<tr>
<td>CBD</td>
<td>Component Based Development</td>
</tr>
<tr>
<td>CDL</td>
<td>Component Definition Language</td>
</tr>
<tr>
<td>CIDL</td>
<td>Component Implementation Definition Language</td>
</tr>
<tr>
<td>CIF</td>
<td>Component Implementation Framework</td>
</tr>
<tr>
<td>CLR</td>
<td>Common Language Runtime</td>
</tr>
<tr>
<td>CLS</td>
<td>Common Language Specification</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DCPS</td>
<td>Data Centric Publish and Subscribe</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Distribution Service</td>
</tr>
<tr>
<td>DLRL</td>
<td>Data-Local Reconstruction Layer</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License (GNU)</td>
</tr>
<tr>
<td>GUID</td>
<td>Globally Unique Identifier</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IL</td>
<td>Intermediate Language</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LGPL</td>
<td>Lesser General Public License (GNU)</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSAL</td>
<td>Operating System Abstraction Layer</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Control(ler)</td>
</tr>
<tr>
<td>RMC</td>
<td>Resource Management Component</td>
</tr>
<tr>
<td>RTE</td>
<td>Run Time Environment</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>SD</td>
<td>Synchronized Data</td>
</tr>
<tr>
<td>SMB</td>
<td>Streaming Memory Buffer</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>SOA</td>
<td>Services Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UD</td>
<td>Unsynchronized Data</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery and Integration</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language (Object Management Group, OMG)</td>
</tr>
<tr>
<td>VCA</td>
<td>Video Content Analyzer</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context and background

ViNotion is a starting organization with a staff consisting of highly skilled specialists in the field of video signal processing and image analysis. They provide innovative automation technology based on intelligent image analysis. Modern image processing techniques in the field of coding and image enhancing depend on the content of the video. Still the true meaning of what is happening in the video is not understood by contemporary systems. Besides information about the video signal such as contrast, intensity or movement in the video image, ViNotion aims at extracting information on a much higher semantic level. For example, a higher semantic level can be tracking and recognizing objects such as people and vehicles. This allows that in many applications the observation by humans is replaced by intelligent systems that can make autonomous decisions or simplify the view presented to humans.

1.2 Problem description

The solutions developed by ViNotion are provided as monolithic software systems. These systems are often computationally expensive and lack the ability to facilitate remote management. We investigate how such monolithic software systems can be broken down into units which can be distributed across a network and allow transparent communication. In order to ensure a uniform approach of developing and distributing a system a framework will be used. A framework can be viewed as an abstraction in which the code providing generic functionality can be selectively parameterized, skipped, replaced or specialized by user code providing specific functionality. In case of ViNotion, the use of a framework would allow a uniform approach of distributing software systems.

1.3 Project goals

The first goal is to identify existing frameworks that allow distribution of software systems across a network. These frameworks are examined within the context of distribution and setting up connections for video streams. If there is no support for streaming between units of distribution, a third party streaming library would need to be integrated.
The second goal is to introduce an abstraction layer that allows the integration of the distribution technology of the selected framework with a third party streaming library. A key part of this integration is maintaining transparency, by using auto-discovery and autonomous stream setups between units of distribution.

Additional goal is to investigate how to improve schemes for setting up streams, allowing resilience against runtime topology changes of the application, like adding or removing units.

1.4 Approach

In order to ensure a structured approach in fulfilling the project goals the following steps are performed:

1. First the requirements that ViNotion imposes on the framework are identified. This goal is achieved by selecting a representative monolithic video processing application developed by ViNotion and investigating which actions are necessary for the distribution. These actions are grouped into different scenarios and are used for identification of the requirements.

2. The next step is to conduct a survey and investigate existing frameworks that can serve as a basis for our framework. In this survey the frameworks are examined with respect to the identified requirements. Based on the importance assigned to requirements by ViNotion a candidate framework is selected.

3. Once a suitable framework is selected, it is analyzed in more detail to identify discrepancies between the offered functionalities and the requirements imposed by ViNotion. For missing functionalities we propose solutions and introduce new concepts. The result is a new framework that serves as a basis for ViNotion to distribute video processing applications. For clarity purposes this desired framework is named ViFramework.

4. Finally, as a proof of concept an application is built by means of the ViFramework to demonstrate its functionalities and to show which requirements have been met. The proof of concept is also used to identify shortcomings of the ViFramework and provide recommendations on how to make improvements.

1.5 Outline

Chapter 2 introduces an application which is a representation for applications developed within ViNotion. By means of this application we identified scenarios for the process of changing a monolithic application to a distributed application. These scenarios resulted in requirements that are used as criteria to select the desired framework. Based on these requirements a high level architecture of the framework is designed for usage within ViNotion.

Chapter 3 describes the results of the survey performed in order to find a suitable existing framework that satisfies the requirements identified in Chapter 2. Based on these requirements a candidate framework is selected. All investigated frameworks, except the candidate framework, are described in more detail in Appendix A. A short overview of the candidate framework and the mapping of its properties to the requirements of the ViFramework is given. Appendix B gives a detailed description of the candidate framework, its architecture, concepts
and properties. In order to investigate if the candidate framework is suitable for streaming video, a feasibility test is performed and the results are presented.

Chapter 4 describes the architecture of a proprietary dedicated streaming library, which serves as an addition to the candidate framework. This library enables functionalities like compression of video streams.

Chapter 5 describes our design that allows the units of distribution to find each other across a network in order to establish communication.

Chapter 6 describes the modifications made to the proprietary dedicated streaming library for integration with the candidate framework. Our design that allows a unit of distribution to establish streaming connections is introduced. Sending and receiving of data by means of buffering and threading is explained.

Chapter 7 elaborates on the design and architecture of a unit of distribution within the ViFramework. This chapter also describes how units of distribution can be used to create a distributed application and explain how data can be sent and received. A proof of concept is introduced, which demonstrates all aspects of the ViFramework.

Finally, Chapter 8 summarizes conclusions and recommendations. It proposes directions for further research and implementation and states some overall concluding remarks.
Chapter 2

Requirements of the ViFramework

First the leading example is introduced that represents the type of applications that are developed and maintained within ViNotion. Having a leading example leads to a better understanding of expectations that ViNotion imposes on the ViFramework. With this leading example we identify scenarios from a system engineers perspective. The goal of these scenarios is to identify requirements and different types of data flows on an architectural level.

2.1 Leading example

The leading example is a virtual fencing application. A virtual fencing application is used for security purposes, like guarding the surroundings of a building with one or more cameras. When an unauthorized person enters a designated restricted area an alarm is triggered. The application is able to detect and classify objects based on visual features. This allows distinguishing between persons and other objects like animals. After an object is classified as a person, the trajectory is considered. Based on the person’s trajectory, the application determines whether this person is authorized to enter the restricted area. E.g. Figure 2.1a shows a person entering the designated area from outside the premises and is classified as an intruder, which results in an alarm. When a person enters the designated area from inside the building, the person is authorized and no alarm is triggered, as displayed in Figure 2.1b. Note that a virtual fence does not necessarily correspond to a physical fence.

![Figure 2.1: Person entering the designated restricted area from different sides.](image)

(a) Unauthorized person. Entering the restricted area by climbing over the physical fence is not allowed.  
(b) Authorized person. Entering the restricted area from inside the building is allowed.
Figure 2.2: The complete virtual fencing system.

The entire virtual fencing system is shown in Figure 2.2. The system consists of a (security) camera and a computer system that runs the virtual fencing application. The virtual fencing application processes the captured images and determines if an alarm should be triggered. When an alarm is triggered, people who are subscribed to the alarm event are notified by means of an SMS containing a snapshot of the intruder.

2.2 *ViFramework* concepts and entities

This section introduces basic concepts and entities envisioned in the *ViFramework*. Figure 2.3 provides an overview how these concepts and entities are related.

2.2.1 Software component

To enable distribution, the virtual fencing application is composed of several software components. The system engineer has to make sure that the components are able to communicate with each other by specifying an interface for each component. Within the context of the *ViFramework* a component can be viewed according to the definition given by Meyer [13]: "A software component can be viewed as a software element (modular unit) satisfying the following conditions:

1. It can be used by other software elements, its 'clients';
2. It possesses an official usage description, which is sufficient for a client author to use it;
3. It is not tied to any fixed set of clients."
Figure 2.3: The architecture of the ViFramework. Note that we distinguish between an application manager, for managing application parameters and a distribution manager, for managing the distribution of the application. This is to emphasize the three different kinds of information flow: 1. application data (black/solid arrows), 2. application control data (red/dashed arrows), 3. distribution control data (blue/dotted arrows).

2.2.2 Docks

A dock can be viewed as a unit of distribution that is a container of components and is. Each software component will be put inside a dock. A dock is able to communicate over the network and contains interfaces to provide network functionality to the containing components. The contained components do not need to know anything about the network and the location of other components. A dock can be viewed as a binary (executable) and specifies the required resources. The idea of using docks as a unit of distribution is derived from the SOFA 2.0 framework [8].
2.2.3 Dock manager

The dock manager is responsible for receiving, installing, removing, starting and stopping a dock. The dock manager is also responsible for setting the dock source and destination addresses for connections, and setting and retrieving the component parameters. A dock manager is designed to be remotely accessible such that the system engineer can remotely control the docks belonging to the dock manager. The dock manager manages all docks that are on the same host.

2.2.4 Hosts

A host is a physical machine in a network. Each host contains one dock manager and can contain zero or more docks. Each dock is managed by the dock manager on the same host.

2.2.5 Dock repository

The dock repository is responsible for maintaining docks. This repository enables maintenance of docks and allows a fast application composition. The idea for a dock repository is derived from the SOFA 2.0 framework [8].

2.2.6 Distribution manager

The distribution manager is responsible for distributing the docks over a given network, managing connections between docks, adding and removing docks from the dock repository. The use of a distribution manager is derived from the SOFA 2.0 framework [8].

2.2.7 Application manager

The application manager is responsible for retrieving, updating and setting the application parameters.

2.2.8 Webserver

The webserver functions as an interface for remote system management. The webserver communicates with the distribution and application manager.

2.2.9 System engineer

The system engineer is able to communicate with the framework through a webserver. The system engineer can upload docks and add them to a host in the network. The system engineer can also extract performance metrics from the distribution manager and is able to initiate a redistribution of the application manually or automatically. Through the application manager the system engineer is able to view and change the values of the application parameters.

2.3 Short overview of architectural scenarios

The system engineer must determine how an application will be composed of software components. These components have to be aggregated into docks. These dock can be distributed over a network and installed on different hosts.
Before deploying docks, a distribution, application and dock manager need to be present in the network. Every host must contain a dock manager that allows the system engineer to manage docks.

Once the system is deployed, the system engineer is able to manage the application from a remote system, change the distribution, set application parameters and retrieve performance and resource metrics. The system engineer can do:

- **Management and control** of the system from a distant location (other than the location of deployment);
- **Load balancing** to distribute the load of the system over several computers;
- **Maintenance** to be able to dynamically reconfigure and upgrade the system;
- **Performance optimization** to remove bottlenecks (as in resource usage) in the system;
- **Physical distribution**, since the system (camera, computers and alarms) will not be deployed on one location;

The system engineer can extend the system, e.g. add an extra camera or a second video content analysis algorithm. An extensive overview of the scenarios is given in Section 2.4.

### 2.4 Architectural scenarios

This section describes architectural scenarios from a system engineer’s perspective. The goal of these scenarios is to identify requirements and different types of data flows, like application data and control data. The scenarios are divided into design of the framework (Scenario 1 to 4) and the actual use of the framework (Scenario 5 to 11).

#### 2.4.1 Scenario 1: Decomposition of the application into components

**Actor**

System engineer.

**Level**

Component.

**Assumption**

The application is available to the programmer.

**Description**

The actor distinguishes three major components for the virtual fencing application,

- Video capturing (VC), for capturing the video stream generated by the camera;
- Video content analyzer (VCA), for object detection and tracking;
Event manager (EM), for determining whether an alarm should be generated and if necessary notifying all subscribed users by means of sending an SMS or e-mail.

The arrows in Figure 2.4 are to indicate what kind of data has to be communicated. The VCA component is further decomposed into two subcomponents, detection and tracking, as shown in Figure 2.5.

Requirements
- The framework should support component based software development;
- (Hierarchical) Nesting/composition of components should be possible;
- Components need to be able to specify provided and required interfaces to be able to communicate with each other, e.g. tracking requires detected objects and a video stream which is provided by the detection component;
- A component should be able to contain attributes (parameters). These parameters are used to control the component and determine its state.

2.4.2 Scenario 2: Aggregate components of an application into docks

Actor
System engineer.

Level
Component.

Assumption
The application is decomposed into components, which communicate through interfaces.
Description

The actor can indicate how components are grouped into docks (see Figure 2.6, which allows distribution of the application over the network in a later stage. For each dock appropriate interfaces are added by the framework. By using docks the application is split up into several binaries (executables) which are atomic.

![Figure 2.6: Mapping the components into Docks.](image)

Requirements

- The framework should be able to distinguish which components are grouped together into docks;
- The framework should be able to specify docks;
- For every dock the appropriate interfaces are added by the framework to enable communication among docks (inter-dock-communication);
- Components of a docks are able to communicate locally via shared memory (intra-dock-communication).

2.4.3 Scenario 3: Initial mapping of docks on hosts

Actor

System engineer.

Level

Network.
Assumption

- The components of the application are aggregated into docks.
- Several hosts are available and have enough resources to run the application.

Description

The actor wants to distribute the different docks over several hosts in a network, see Figure 2.7. To do this docks have to be able to communicate over the network. A host can contain one or more docks.

![Figure 2.7: Distribution of docks over several hosts in the network.](image)

Requirements

- The framework should support network functionality by means of a network communication protocol, like TCP/IP;
- For every dock the appropriate network interfaces are added by the framework to enable communication among docks over the network.

2.4.4 Scenario 4: Host preparation

Actor

System engineer.

Level

Network.

Assumption

Four hosts are available and their resources are known.

Description

Four hosts are available on which the example application will be deployed. One host will be used for the application and distribution manager (Host D), which are responsible for the distribution and control of the docks. On the hosts that will run docks one dock manager
will be installed and started, which is responsible for managing the docks on its host (Hosts A,B,C). See Figure 2.8 for a graphical overview.

Figure 2.8: Distribution of the Managers several hosts in the network.

Requirements

- It should be possible to install a dock manager on each server that will host docks;
- The distribution manager knows which hosts are part of the network.

2.4.5 Scenario 5: Deploying docks

Actor

System engineer.

Level

Network.

Assumption

- The hosts in the network are prepared: A distribution manager, application manager are installed and on each hosts a dock manager is installed.
- Communication between distribution manager and dock managers is possible.

Description

The actor is able to add or remove a dock with the use of the distribution manager and the dock repository. When the actor adds a dock to the network, the (precompiled) dock is retrieved from the dock repository and sent to the appropriate dock manager. The dock manager will install the received dock on its host, see Figure 2.9. The distribution manager maintains the locations of the installed docks. For Example, the distribution manager provides Dock 3 to the dock manager of Host C and tells the dock manager to install Dock 3 onto its host. When a dock is being replaced by a new one (with the same interfaces), the current
Figure 2.9: Add a dock to a host using the distribution manager, the dock repository and the dock manager.

state of the dock will be lost and the application has to be put on hold and restarted when the new dock has been initialized and ready to run. The docks added to the network are also stored in the repository, this allows the actor to deploy docks from the repository to the network.

Requirements

- The distribution manager is able to store the dock host-locations;
- The distribution manager has a list of all available hosts;
- The distribution manager ordains all available docks, by means of the dock repository.

2.4.6 Scenario 6: Real-time dock performance extraction

Actor

Distribution manager (is automated by the system).

Level

System.

Assumption

The distribution manager and the dock managers are able to communicate control data.
Description

The distribution manager retrieves the average performance metrics like CPU load and memory usage periodically from the dock managers and stores these in the repository, see Figure 2.10. The period can be set by the programmer/system architect. These metrics can be used by the programmer/system architect for the development of future applications or for regrouping components over the docks. These metrics could also be used for redistribution of the docks over the network (see also Scenario 9 and 10).

Figure 2.10: Retrieve performance metrics from the Dock managers.

Requirements

- The dock manager can measure performance metrics like average CPU load and memory usage of its docks;
- The dock manager is able to communicate with the operating system;
- The distribution manager periodically retrieves the performance metrics from the dock managers;
- The distribution manager maintains the performance metrics.

2.4.7 Scenario 7: Manage connections between docks

Actor

System engineer.

Level

Network.
Assumption
- The hosts in the network are prepared and the docks are deployed on the hosts.
- The distribution manager is able to communicate with the dock managers.

Description
The actor is able to maintain (data)connections between docks with the use of the distribution manager, as shown in Figure 2.11. The connections are stored by the distribution manager and are used by the docks to find each other. E.g. Dock 2 has to communicate with Dock 3. This connection is stored by the distribution manager. When Dock 3 is deployed its location (IP-address) is registered within the distribution manager. Since Dock 2 needs to be connected to Dock 3, Dock 2 will be notified by the distribution manager through the dock manager about the (updated) location of the Dock 3. Thus the distribution manager provides the host-location of the different docks as well as the connections between the different docks. Once docks are able to locate each other, connections between the docks can be established and data can be communicated. Depending on the application multiple connections between docks can be made to communicate data.

Requirements
- The dock managers know the location of the distribution manager;

Figure 2.11: Maintains (data)connections between docks with the use of the distribution manager.
• The distribution manager knows the locations of all the active docks in the application;
• The distribution manager is able to register connections between docks.

2.4.8 Scenario 8: Initiate a redistribution of the application

Actor
System architect.

Assumption
The application has been deployed on site

Description
The actor can initiate a redistribution of the system in three ways:
• "I want this distribution of the application", where the actor provides a new schedule to the distribution manager (see Scenario 9)
• "I want a new distribution of the application", where the system provides a new distribution on the basis of the performance measurements (see Scenario 10)
• "I want a new distribution every once in a while", where the actor can specify the period for which the distribution manager should look for a better distribution (if available), according to the continuously retrieved performance measurements (see Scenario 10)

2.4.9 Scenario 9: Configure the distribution of the application manually

Actor
System engineer.

Level
Network.

Assumption
The application has been deployed on site.

Description
Since the application has been deployed on site, the actor wants to be able to configure the application remotely, without having to access the hosts physically, as shown in Figure 2.12. To do this a webserver is installed, which communicates to the distribution manager. The application can be configured by means of the distribution manager. The actor has the possibility to control the distribution manager. This is done by means of a user interface which is able to connect with the distribution manager through a webserver. Once connected the actor is able to manage the mapping of the application, i.e. the locations of the docks and the connections between them. The user is also able to upload docks to the repository and install them on a host.
Figure 2.12: Remotely configure the application on site through a webserver.

Requirements

- The location of the webserver must be known and accessible by the user;
- The location of the distribution manager must be known by and accessible via the webserver;
- The distribution manager must be able to access the dock repository.

2.4.10 Scenario 10: Configure the distribution of the application automatically

Actor

System and system architect.

Level

Network.

Assumption

The application has been deployed on site.

Description

The actor can initiate an automatic rescheduling of the docks over the available hosts. The distribution manager will (re)schedule docks on the available hosts based on the resource information of the docks. The distribution manager periodically retrieves updates from the
dock manager about the performance metrics of its docks (see Scenario 6) and the availability on the hosts is known.

This scenario implies that the distribution manager should have a component which is responsible for dynamic scheduling of docks on the available hosts. Another implication is the decision making, how do CPU load, memory usage and bandwidth relate in the current context and result in near optimal decision?

Requirements

- A scheduling policy needs to be defined, defining how and when to schedule;
- An upper bound on time needs to be defined for finding an approximation of the optimal dock scheduling on hosts;
- Initial performance metrics on CPU load and memory usage need to be available, either by testing, or real-time extraction;
- The distribution manager has a component which is responsible for dynamic scheduling of docks on the available hosts.

2.4.11 Scenario 11: Get/set application parameters

Actor

System engineer.

Level

Network and component.

Assumption

The application has been deployed.

Description

The actor must be able to get/set the parameters of the application by starting up a browser and connecting to the webserver, see Figure 2.13. The webserver retrieves the application parameters from the application manager. The actor gets or sets the parameters. E.g. the actor wants to set the designated area and manage the subscription for SMS alerts, which is related to the Event Manager (EM) of the Virtual Fencing application.

Requirements

- A component must have a control interface through which its parameters can be get or set;
- An application manager is part of the framework which facilitates retrieving, changing and updating the application parameters;
- A user is able to connect to the webserver;
The webserver is able to connect to and communicate with the application manager.

2.5 Failure Scenarios

2.5.1 Fault model

All architectural scenarios are executed without occurrence of any failures. In order to increase the framework’s robustness a fault model is introduced where we differentiate between the following types of errors: connectivity, availability and maintainability errors. Table 2.1 shows a more fine-grained decomposition of these types of errors.

2.5.2 Failure scenario 1: Data connection time out

Actor

System.

Level

Network and dock.

Assumption

- The application has been deployed and dock 2 (in the example) is unreachable;
<table>
<thead>
<tr>
<th>Type of error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity errors</td>
<td></td>
</tr>
<tr>
<td>- Data connection failure</td>
<td>A connection used for data transport fails</td>
</tr>
<tr>
<td>- Control connection failure</td>
<td>A connection used for control transport fails</td>
</tr>
<tr>
<td>Availability errors</td>
<td></td>
</tr>
<tr>
<td>- Dock unavailable</td>
<td>A dock is not responsive</td>
</tr>
<tr>
<td>- Dock manager unavailable (host down)</td>
<td>A dock manager is not responsive</td>
</tr>
<tr>
<td>Maintainability errors</td>
<td></td>
</tr>
<tr>
<td>- Install/remove failure</td>
<td>A dock manager is unable to remove or install a dock</td>
</tr>
<tr>
<td>- Add/retrieve from repository failure</td>
<td>The distribution manager is unable to store/retrieve a dock from the repository</td>
</tr>
</tbody>
</table>

Table 2.1: Fault model showing connectivity, availability and maintainability errors.

- All dock managers are reachable for the distribution manager;
- The dock managers know the location of the distribution manager.

Cause(s)
- Dock unavailable error;
- Data connection error.

Description
When communication between two docks on different hosts fails, it can be either caused by a broken connection or the unavailability of a dock. As an example (see also Figure 2.14), Dock 1 sends a message to Dock 2 on Host B but gets back a time out. Dock 1 then reports this to its local Dock Manager, which in turn reports to the distribution manager that Dock 2 is unreachable. The distribution manager sends a request for a restart of Dock 2 to the Dock manager of Host B. The Dock manager on Host B restarts Dock 2 and the application can continue running. If the destination address is incorrect the dock manager will request the (new) destination address and retries to establish a connection.

Note that when a dock is restarted its current state is lost. A dock is restarted with parameters which were initially set by the application manager.

Catch
If this scenario fails, thus a restart of dock 2 does not resolve the problem, an alert will be sent to any subscribed users who are responsible for managing the application, see the next failure scenario.
Figure 2.14: Failure scenario: data connection timeout, which indicates a broken connection or the unavailability of a dock.

Requirements

- Each host contains a dock manager;
- The distribution manager knows the locations of all deployed docks in the application;
- User can subscribe to application failure events.

2.5.3 Failure scenario 2: Host is unavailable

Actor

System.

Level

Network.

Assumption

- Initially a time out is observed by a certain dock or the distribution manager;
- When the dock manager is non-responsive we assume the host is down.
**Cause(s)**
- Dock manager unavailable (host down);
- Control connection failure.

**Description**
After Dock 1 was unable to connect to Dock 2, the distribution manager tries to notify the Dock Manager on Host B that Dock 2 is non-responsive. In this scenario the Dock Manager is also non-responsive. In such a case we assume that Host B is down and the list of available hosts is updated by the distribution manager. See also Figure 2.15 for a graphical description of this failure scenario. At this moment we suggest 2 options:

1. The distribution Manager sends an alert to the webserver and the webserver notifies subscribed users who are responsible for maintenance on the application.

2. The distribution Manager also sends an alert to the subscribed users, as in 1), and initiates rescheduling as described in Scenario 9.

![Figure 2.15: Failure scenario: host is unavailable.](image)

**Requirements**
- The webserver needs to maintain a list of users which are notified in case a host is down;
- A dock manager must be able to calculate resource consumptions like, CPU load, memory usage and bandwidth;
- Each dock stored in the repository should also contain average performance metrics to enable dynamic scheduling of docks.
2.5.4 Failure scenario 3: Unsuccessful dock remove/install

Actor
System.

Level
Dock and network.

Assumption
- The dock manager is reachable and operating correctly;
- The distribution manager is reachable and operating correctly.

Cause(s)
Remove/install failure.

Description
If a dock cannot be successfully installed or removed, the webserver should be notified via the distribution manager. The webserver should notify any subscribed users to failure events.

Requirements
The webserver should contain an event handler for dealing with failure events.

2.5.5 Failure scenario 4: Unsuccessful dock retrieval and insertion

Actor
System.

Level
Dock and network.

Assumption
The distribution manager is reachable and operating correctly.

Cause(s)
Add/retrieve from repository failure.

Description
If a dock cannot be successfully inserted or retrieved to or from the dock repository, the distribution manager should notify the webserver which on its turn should notify any subscribed users to failure events.
Requirements

- The webserver should contain an event handler for dealing with failure events;
- The distribution manager should be able to generate exception with regards to dock insertion and retrieval to or from the repository.

2.6 Requirements

Using the scenarios from Section 2.4 the requirements for the ViFramework are identified and presented in this section. ViNotion also imposes pragmatic requirements on the ViFramework.

Based on these requirements, a suitable existing framework is selected that serves as a starting point for the ViFramework. The discrepancies between the selected framework and the ViFramework are determined and the selected framework is extended accordingly (see Chapter 3).

2.6.1 Pragmatic requirements

ViNotion imposes the following requirements on the ViFramework:

- Open source, with a license that allows us to adapt a framework;
- C++ compatible, all programming activities within ViNotion are done in C++;
- Multi platform, the framework must be compatible with Linux as well as Windows;
- Maintainability, the framework must have complete documentation. The framework must be supported by an active community, that develops and releases new versions of the framework and provides support.

2.6.2 Composition requirements

The ViFramework has the following requirements according to the scenarios that are described in Section 2.4:

- Component based software development, the framework is able to specify components and store them in a repository for reuse;
- Hierarchical nesting of components, the framework allows hierarchical nesting of components in such a way that a component can contain several subcomponents;
- Provide and require interfaces, per component these interfaces are defined for local communicating purposes. Each component is able to provide and/or require functionalities;
- Component parameters, each component has parameters that specify the behavior of the component and which can be accessed from the outside;
- Component grouping into docks, the framework has the ability to group several components into a dock that serves as a unit of distribution;
- Dock specification, the dock facilitates network functionality (inter-dock-communication);
2.6.3 Deployment and distribution requirements

- Distribution manager, the framework contains a distribution manager;

- Application Manager, the framework contains an application manager;

- Dock managers, the framework is able to add dock managers to the application in order to manage docks running on one host;

- Communication, the Dock manager, Application manager and distribution manager are able to connect with each other and communicate control and application data;

- Web server, is used to control the distribution and application manager. From a system engineers perspective the distribution and application manager are services offered by the ViFramework.

2.6.3.1 The distribution manager requirements

- Locations, the distribution manager distributes docks and keeps track of the installed docks. The location is the address of a host in the network;

- Connections, the distribution manager manages connections between installed docks. This way when a location of a dock is changed, the connected docks can be notified;

- Dock repository, the distribution manager is able to store complete docks in a repository for installation on hosts. The advantage of a repository is that the system engineer can use docks without uploading via the webserver;

- Resources, the distribution manager is able to query what resources are available on each host;

- Performance monitoring, the distribution manager monitors performance metrics of the application on dock level (e.g. CPU cycles and memory usage);

- Redistribution, the distribution manager is able to automatically (or by initiative from the systems engineer) redistribute the application over the available hosts, based on the performance monitoring results;

- Provide and require interfaces, the distribution manager has network interfaces (with IP-functionality) to communicate over the network.

2.6.3.2 The application manager requirements

- Parameters, the application manager is able to retrieve/set the parameters from the different docks installed in the network;

- Provide and require interfaces, the application manager has network interfaces (with IP-functionality) to communicate over the network.
2.6.3.3 The dock manager requirements

- *Manage docks*, the dock manager is able to receive, install, run, pause, stop and remove docks on the host;

- *Manager communication*, the dock manager is able to communicate over the network with the Distribution and Application manager;

- *Resource information*, the dock manager is able to query the host what resources are available;

- *Performance measurements*, the dock manager is able to calculate the resource usage of the docks installed on its host;

- *OS communication*, the dock manager is be able to communicate with the OS of the host for resource information and performance measurements.

2.6.4 Requirements priority

Since probably no existing framework will satisfy all of the requirements, a choice for a framework should be based on certain criteria. In order to select a suitable framework, the importance of each requirement is identified. This importance is represented using priorities. Tables 2.2, 2.3 and 2.4 show the priorities per requirement, as given by ViNotion, where 1 is the highest priority and 5 is the lowest priority. Also sub-priorities are used where e.g. 2.3 means a requirement has priority 2 and within this priority it has priority 3. So a requirement with priority 2.3 has a higher priority than one with priority 3, but a lower priority than another sub-requirement with priority 2.1. When selecting a suitable framework the pragmatic requirements are the most important.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open source</td>
<td>1</td>
</tr>
<tr>
<td>C++ compatible</td>
<td>1</td>
</tr>
<tr>
<td>Multiplatform</td>
<td>1</td>
</tr>
<tr>
<td>Maintainability</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 2.2: Pragmatic requirements priorities*

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component based software development</td>
<td>2</td>
</tr>
<tr>
<td>Component grouping into docks</td>
<td>1</td>
</tr>
<tr>
<td>Dock specification</td>
<td>1</td>
</tr>
<tr>
<td>Network functionality</td>
<td>1</td>
</tr>
<tr>
<td>Provide and Require interfaces</td>
<td>1</td>
</tr>
<tr>
<td>Hierarchical nesting of components</td>
<td>3</td>
</tr>
</tbody>
</table>

*Table 2.3: Composition requirements priorities*
<table>
<thead>
<tr>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution manager</td>
<td>2</td>
</tr>
<tr>
<td>- Locations</td>
<td>2.1</td>
</tr>
<tr>
<td>- Connections</td>
<td>2.1</td>
</tr>
<tr>
<td>- Provide and require interfaces</td>
<td>2.1</td>
</tr>
<tr>
<td>- Dock repository</td>
<td>2.2</td>
</tr>
<tr>
<td>- Resources</td>
<td>2.3</td>
</tr>
<tr>
<td>- Performance</td>
<td>2.4</td>
</tr>
<tr>
<td>- Redistribution</td>
<td>2.5</td>
</tr>
<tr>
<td>Application manager</td>
<td>2</td>
</tr>
<tr>
<td>- Parameters</td>
<td>2.1</td>
</tr>
<tr>
<td>- Provide and require interfaces</td>
<td>2.1</td>
</tr>
<tr>
<td>Dock manager</td>
<td>3</td>
</tr>
<tr>
<td>- Manage docks</td>
<td>3.1</td>
</tr>
<tr>
<td>- Manager communication</td>
<td>3.1</td>
</tr>
<tr>
<td>- Resource information</td>
<td>3.3</td>
</tr>
<tr>
<td>- Performance measurements</td>
<td>3.4</td>
</tr>
<tr>
<td>- OS communication</td>
<td>3.3</td>
</tr>
<tr>
<td>Communication</td>
<td>2</td>
</tr>
<tr>
<td>Web server</td>
<td>3</td>
</tr>
</tbody>
</table>

*Table 2.4: Deployment and distribution requirements priorities*
Chapter 3

Framework survey

Based on the requirements identified by means of the scenarios a survey has been conducted to find a suitable existing framework that can serve as a basis of the ViFramework. The conducted survey, the findings and the result are presented in this chapter. The frameworks are classified on the semantics and syntax to investigate if a framework suits the development techniques and methods of ViNotion. Based on these properties and requirements a candidate framework is selected. All investigated frameworks, except the selected one, are described in more detail in Appendix A.

3.1 Introduction

Based on the requirements identified by means of the scenarios in Chapter 2, the framework can be decomposed into two elements. The first element is designing and implementing software by means of components, Component Based Development (CBD), which diverges from the traditional monolithic software designs. This enables reuse of prebuilt software units and thus reducing production cost and time-to-market. The other element is the middleware aspect. The middleware is responsible for disseminating/communicating data to other docks which are distributed over a network.

![Layer View](image)

Figure 3.1: A layer view of an application running on top of middleware

Figure 3.1 shows an abstract layer view of an application running on top of middleware. The application layer can be constructed by means of CBD. How software is constructed by means of CBD depends on the underlying component model. A component model defines what components are, how they should be used, assembled, composed and deployed. Lau et al [12] distinguishes two major categories, a model where components are objects, as in Object-
Oriented Programming (OOP) and a model where components are architectural units, as in software architectures.

3.2 Survey

With this survey we try to find a suitable framework, which satisfies the requirements as defined in Chapter 2. This component/service based open source framework serves as a basis of the ViFramework. The investigated frameworks are described in more detail in Appendix A. Using the requirements identified in Chapter 2, we have created a table which provides an overview of the found frameworks and their properties, see Table 3.1. By looking at the component model of the framework (as in Figure 2.3) we tried to gain more insight in the differences between the several frameworks.

Lau et al [12] state that a component model should define the following:

- **Semantics**, what components are meant to be;
- **Syntax**, how are components defined, constructed and represented;
- **Composition**, how are components composed or assembled.

Lau et al [12] created a taxonomy based on these elements of a component model. Considering the component semantics, they classify three categories:

1. Models in which components are modeled by classes (Cls);
2. Models in which components are objects (Obj), where objects can be viewed as runtime entities;
3. Models in which components are architectural units (AU).

In order to be able to define components, the component model should provide a component definition language, which may differ from the implementation language. Based on component syntax they classified three categories:

1. Components are defined by object-oriented programming languages;
2. Components are defined by means of IDL (Interface Definition Language), and implemented by means of programming languages with IDL mappings. The interfaces of the component are specified by means of the IDL language;
3. Components are defined by an ADL (Architectural Definition Language), where the ADL defines the architectural level of the application on a component level.

Lau et al [12] created an additional taxonomy based on the idealized life cycle of a component. We did not focus on this idealized life cycle and have only looked at composition at design time, called component nesting. We also included a criteria for classification whether a framework is component oriented or service oriented. See Table 3.1 and legend below for the overview.
3.3 Selected candidate and motivation

Table 3.1 reveals that none of the considered frameworks satisfy all of the requirements. The frameworks that are not open source, have no support for the C/C++ programming language and are not compatible with both Windows and Linux are not considered to be candidates for the ViNotion framework. Therefore the following candidates are not considered: .NET, Web services, JavaBeans, KobrA, CANTATA, Robocop, openCCM, SOFA2, PECOS, Pin, Koala. Some of these frameworks however have good support for the other requirements, such as SOFA2. The concepts used by these frameworks will be studied and used to extend
the selected framework with the requirements that are not satisfied.

After eliminating candidates based on the key requirements, the potential number of frameworks has been reduced significantly to only three frameworks:

- **GStreamer** is widely used for streaming applications, such as audio and video, but it has some unknown requirements, due to insufficient documentation, no built-in network functionality and is specialized in building music players focused on the use of codecs and creation of user interfaces.

- **Fractal** has several implementations and is supported by an active community of developers (France Telecom), but very few implementations provide support for network functionality and distribution. Comet is such an implementation, but does not have support for Windows/Linux. Currently only the C programming language is supported, support for developing components in C++ is still experimental.

- **OpenDDS** has support for the C++ programming language, which means that no wrapper has to be made to port C-code to C++. It also has build-in network functionality and has a distribution manager.

For a more detailed overview of the available framework we refer to Appendix A.

By means of the identified requirements and their priorities, we recommend to use **OpenDDS** as the basis of the **ViNotion** framework. Since **OpenDDS** supports the C++ programming language and the network functionality requirements, which both have high priority. **OpenDDS** allows for specification of services/components by means of IDL interfaces. The interfaces are mapped to generated implementation code. **OpenDDS** components can be viewed as object like runtime entities, these objects are constructed by means of C++ classes. **OpenDDS** does not specify the support for streaming video data, as opposed to GStreamer which has good support for streaming video data, but does not facilitate network functionality. Currently **ViNotion** designs software by means of OOP, which has a good correspondence with **OpenDDS** and contributes to good integration of the framework.

### 3.4 Data Distribution Service short overview

In this section we provide a short overview of how DDS enables communication. We introduce concepts and entities and explain how they are related.

The DDS specification is decomposed in two separate layers:

1. the Data Centric Publish-Subscribe (DCPS) layer, responsible for disseminating data to other DDS applications;

2. the Data Local Reconstruction Layer (DLRL), specifies how an application can access the interfaces of DCPS data fields through its own object oriented programming classes.

**OpenDDS** only implements the DCPS layer. DCPS is comprised of the following entities:

- **Domain**, a domain is used to bind applications together for communication;
- **Domain Participant**, an entry object that allows an application access to one domain. A domain participant can only reside in one domain, if an application developer wishes to access multiple domain from a single application multiple domain participants are required;

- **Data Writer**, a primary access for an application to publish data into a domain;

- **Publisher**, a container entity for grouping Data Writers, allow to specify a single Quality of Service (QoS) policy applying to all Data Writers;

- **Data Reader**, a primary access for an application to receive data from a domain;

- **Subscriber**, a container entity for grouping Data Readers, allow to specify a single Quality of Service (QoS) policy applying to all Data Writers;

- **Topic**, a connection point between Publishers and Subscribers. Topics of a Publisher and Subscriber must match to allow communication.

![Figure 3.2: DDS components, taken from Farabaugh et al [17]](image)

Figure 3.2 shows how the concepts/entities relate to each other. Data samples are sent and received to/from the domain. Publishers and Subscribers manage Data Writers and Data Readers. Note that a Publisher/Subscriber can have ownership over several Data Writers/Readers. Data Writers/Readers are associated to each other by means of Topics.

Figure 3.2 shows three different types of data streams (1, 2 and 3). Subscribers which have subscribed to topic 1 will not receive any information about topics 2 or 3. This allows for a nice separation of functionality within a domain.

### 3.5 Streaming video by means of OpenDDS

When communicating between nodes, OpenDDS marshals the data samples at the publisher side and unmarshals the data samples at the subscriber side. Marshalling (similar to serialization) is the process of transforming the memory representation of an object to a data format suitable for storage or transmission. Marshalling data allows generic interfacing between the network and the Datawriter/Datareader at the cost of overhead in sample size.

This overhead causes additional load on the network, which could cause congestion in the network resulting in back pressure on application. In this section we investigate if this
overhead causes problems when streaming data samples by means of an experiment. The goal of this experiment is twofold:

- Acquire hands-on experience;
- Investigate if OpenDDS is suitable for handling video streams, with respect to marshalling overhead and compression.

3.5.1 Experiment

3.5.1.1 Hardware and software specifications

The available bandwidth between the sender and receiving application is 1 gigabit/s. The experiment will be conducted on the hardware platforms as specified in Table 3.2. The tools and libraries used are specified in Table 3.3.

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Centrino Core 2 duo @ 1.7 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>2048 MB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows XP Professional SP3</td>
</tr>
<tr>
<td>Network connection</td>
<td>1.0 Gigabit</td>
</tr>
<tr>
<td>Camera</td>
<td>Logitech QuickCam Pro 5000</td>
</tr>
<tr>
<td>Camera driver</td>
<td>Version 9.5.0</td>
</tr>
</tbody>
</table>

Table 3.2: Platform specifications

<table>
<thead>
<tr>
<th>Tool/Library</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS Visual Studio</td>
<td>2008 professional edition version 9.0.21022.8 RTM</td>
</tr>
<tr>
<td>OpenDDS</td>
<td>Version 2.0.1</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Pre1.1</td>
</tr>
</tbody>
</table>

Table 3.3: Tool and library versions

3.5.1.2 Application design

We use two applications, a sending and receiving application (see Figure 3.3). The sending application captures frames from a webcam, copies the image data to samples and publishes these samples. The receiving application displays the frames from the stream.

The video data stream between the applications has a resolution of 320x240 and is captured from webcam with 15 frames per second.
3.5.1.3 Experimental setup

In order to investigate if marshalling video data introduces additional bandwidth consumption, we measure the bandwidth consumption by means of Windows Task Manager. To determine the overhead we compare the result of the Windows Task Manager to theoretical bandwidth consumption that is calculated by the following formula:

\[
Utilization = \frac{width \cdot height \cdot pixelsize \cdot framerate}{bandwidth} \cdot 100
\] (3.1)

The video data is copied into data samples without any compression technique. Using compression would result in a much more complicated formula making it much more difficult in estimating the theoretical bandwidth.

Some considerations have to be taken into account when using Windows Task Manager:

- The accuracy of the bandwidth consumption is unknown, the refresh rate is set to 2 Hz;
- The bandwidth consumption of the entire system is measured, i.e. bandwidth consumption of background processes are also accumulated in the result.

3.5.2 Findings

Figure 3.4 shows that the bandwidth consumption measured with Windows Task Manager when using a single stream of 320x240 pixels (24 bits per pixel) over the network is about 3%. This coincides with the results of the formula, since it yields a utilization of \(\frac{320 \cdot 240 \cdot 24 \cdot 15}{10^9} \cdot 100 \approx \)
2, 76%. The difference in values could be explained by the precision of Windows Task Manager and additional overhead in size due to marshalling of the video data.

3.5.3 Findings

The overhead of marshalling data is negligible, approximately 0.3% including possible network load induced by background processes. Thus the overhead of marshalling can be considered low when streaming video data.

However, due to practical experiences, we gained insight in the use of data samples. Data samples are defined at compile time and can only contain static elements, i.e. dynamic data types like dynamic arrays cannot be used. This complicates the use of compression and becomes very important when scaling the number of video data streams. Figure 3.5 shows how the bandwidth consumption increases.

A solution would be to define several data samples with different sizes, e.g. 10, 20 and 30 KB. A compression algorithm can then select a data sample with the highest utilization. This is however not a very elegant solution, since it introduces additional overhead in selecting the appropriate data sample. A better solution would be to use a sample which could vary in
size. This is not possible without making adaptations to the OpenDDS framework. As an alternative solution we explore the option of using a proprietary dedicated streaming library. Although not suitable for streaming video data, OpenDDS is very suitable for communicating control data. Separate topics can be used for different control data flows.
Chapter 4

Streaming library

ViNotion provides a proprietary streaming library for handling video streams, called \textit{lib-ViStreaming}. The architecture of this library is described in more detail. The structure of the library is split up into two parts: a server and a client. The library allows to setup multiple video streams between a client and a server. Furthermore these video streams can be compressed based on the available bandwidth. Meta data can be sent and synchronized with the video stream.

4.1 Architecture (design)

This section will explain the architecture of the library, its main relations and components. The library can be divided into two main parts: server and client. An overview of the library’s architecture is shown in Figure 4.1.

4.1.1 Server side

The server side consists of three main types of objects:

- \textit{Data exchangers}, responsible for sending/receiving data and information over the network.
- \textit{Managers}, taking care of sending/receiving data to/from particular clients.
- \textit{Interfaces}, in charge of easing the data exchange between the server application and the library.

4.1.1.1 Data exchangers

\textbf{OutputStream}: this object is able to encapsulate any kind of data into RTP packets and send them over the network.

\textbf{VideoOutputStream}: this object is linked to a CodecProfile, which deals with the compression of video frames. Once compressed, the frames are then encapsulated into RTP packets and send over the network.

\textbf{EventChannel}: this object is in charge of sending/receiving events over the network using a TCP connection.
Figure 4.1: Global architecture of the libViStreaming library, taken from the libViStreaming documentation.

**UDChannel:** this object is in charge of sending/receiving any kind of data. When used to send data, it is in charge of encapsulating the data into TCP packets and sending them over the network. When used to receive data, it is in charge of reconstructing the data.

### 4.1.1.2 Managers

**ServerSession:** this object represents one client connected to the server. This way it is easy for the server to communicate with a particular client.

**ServerManager:** the top object of the server. It contains all the main functionalities of the server (connection, stream). It manages the connections with the different ServerSessions.

### 4.1.1.3 Interfaces

**UDListener:** in charge of the data communication (sending/receiving) between the server application and the library.
**EventListener:** in charge of the events communication (sending/receiving) between the server application and the library.

### 4.1.2 Client side

As for the server side, the client side consists of three main types of objects:

- **Data exchangers**, responsible for sending/receiving data and information over the network.
- **Managers**, taking care of sending/receiving data to/from particular servers.
- **Interfaces**, in charge of easing the data exchange between the client application and the library.

#### 4.1.2.1 Data exchangers

**InputStream:** this object is able to reconstruct the data sent by an OutputStream via RTP packets and pass them to the synchronizer for further treatment.

**VideoInputStream:** this object is linked to a CodecProfile who deals with the decompression of video frames. Once decompressed, the raw frames are then passed to the synchronizer.

**EventChannel:** this object is in charge of sending/receiving events over the network using a TCP connection.

**UDChannel:** this object is in charge of sending/receiving any kind of data. When used to send data, it is in charge of encapsulating the data into TCP packets and sending them over the network. When used as to receive data, it is in charge of reconstructing the data.

#### 4.1.2.2 Managers

**ClientSession:** this object represents the connection of the client with one particular server. It manages both the input and output from/to one server.

**ClientManager:** the top object of the client. It contains all the main functionalities of the client (connection to a stream, sending events).

**Synchronizer:** this object is in charge of synchronizing the streams received by the client. The synchronization is made on two different levels: it synchronizes the video streams with their associated data and it synchronizes between the video streams.

Once synchronized, the streams are sent to the client manager who pushes them toward the VideoDataListener in order to make them available to the application.
4.1.2.3 Interfaces

**UDListener:** in charge of the unsynchronized data communication (sending/receiving) between the client application and the library;

**EventListener:** in charge of the events communication (sending/receiving) between the client application and the library;

**VideoDataListener:** in charge of transmitting the received frames and their associated data from the library to the client application. This is also where the frame/data drops are notified to the developer.

4.1.3 Development view

The client and the server side of the streaming library are represented in Figure 4.2. The server side compresses video frames, streams the video content, sends synchronized data (SD) and sends/receives both unsynchronized data (UD) and events. The client side receives the video streams as well as the SD streams, decompresses video frames, synchronizes the video streams with SD streams and sends/receives both UD and events.

<table>
<thead>
<tr>
<th>Client Side Layers</th>
<th>Server Side Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Application</td>
<td>Server Application</td>
</tr>
<tr>
<td>ClientManager</td>
<td>ServerManager</td>
</tr>
<tr>
<td>Synchronizer</td>
<td></td>
</tr>
<tr>
<td>CodecProfile</td>
<td></td>
</tr>
<tr>
<td>ClientSession</td>
<td>ServerSession</td>
</tr>
<tr>
<td>InputStream,</td>
<td>Channel, StreamChannel,</td>
</tr>
<tr>
<td>VideoInputStream</td>
<td>BidirectionalChannel,</td>
</tr>
<tr>
<td></td>
<td>EventChannel, UDChannel</td>
</tr>
<tr>
<td>RTP</td>
<td>TCP</td>
</tr>
<tr>
<td></td>
<td>OutputStream,</td>
</tr>
<tr>
<td></td>
<td>VideoOutputStream</td>
</tr>
</tbody>
</table>

*Figure 4.2: Layering of the libViStreaming library.*

**Server side layers** The server side consists of five different layers.

- **Server Application**, this layer contains the server application and is responsible for sending and receiving information;
- **Server Manager**, this layer contains the management classes for the server;
- **Codec profile**, this layer contains the classes that realize the actual compression and decompression of the video content;
- **Server session**, this layer is for the classes that identify a connection to a client and for the logical grouping of channels;
- **Channels**, the channels layer contains the classes that actually make the connection between the higher abstraction (output streams) and lower level classes that deal with the network connections (RTP/TCP).

**Client side layers** Many of the client side layers can be found on the server side as well. The only difference is that the client side contains the Synchronization layer in addition which incorporates the classes that deal with synchronization issues.

### 4.1.4 Subsystem flow

This section describes the dynamic flow of data received from the network through the layers described in Section 4.1.3. There are three different types of data with a flow on the server and a flow on the client. The streaming (synchronized and video) data makes use of RTP as a transport protocol. The non-streaming data (events, unsynchronized data) makes use of TCP to transport this data.

**Synchronized data** on the server side goes through the Application, ServerManager, ServerSession (skipping the CodecProfile) and the OutputStream. On the client side, the synchronized data goes first through the InputStream and ClientSession (again skipping the CodecProfile) and then through the Synchronizer, ClientManager and to the ClientApplication.

**Video data** on the server side goes through all layers (in the TransmitStream layer it goes specifically through a VideoOutputStream). On the client side, the video data again goes through all layers (VideoInputStream in the ReceiveStream layer).

**Non-streaming data** on the server side goes through the ServerApplication, ServerManager and also through the ServerSession layer (skipping the CodecProfile layer) to arrive at the EventChannel or UDChannel (depending on the type of the data). This layer passes the data through the TCP layer for sending the data to the client. On the client side, the non-streaming data goes from the respective channel through the ClientSession, to the ClientManager and the ClientApplication, skipping the Synchronizer and the CodecProfile layers. For sending non-streaming data from client to server, it goes through the same layers, but in the other direction.

### 4.2 Mapping to requirements

#### 4.2.1 Streaming facilities

`libViStreaming` provides streaming facilities for streaming video and associated data. The library is able to compress and decompress video data using different codec profiles.

#### 4.2.2 Inter-dock communication

The `libViStreaming` library provides interfaces for communication among docks. Managers and sessions are used to manage the connections between docks used for communication.
4.2.3 Network functionality

Using the *libViStreaming* library provides network functionality to the *ViFramework*. This network functionality is supported within *libViStreaming* using different network communication protocols. TCP/IP is used for unsynchronized data and events communication, while RTP is used for streaming video and data communication.
Chapter 5

Auto discovery of docks

This chapter explains how docks use auto discovery to resolve an application topology defined by means of using abstract identifiers. This allows the system engineer to focus on designing an application topology and to abstract from complicated handshakes and the use of manual connection setups.

5.1 Client Server connection setup

Let’s assume that a systems engineer has grouped components into three docks, see Scenario 2 in Chapter 2, and wants to deploy these docks on separate hardware platforms which are part of a network, see Figure 5.1.

![Figure 5.1: Grouping software components into docks](image)

In Client-Server communication proposed in Chapter 4, a server is passive and awaits an active request from the client to establish a connection. In this particular example, Dock 2 will be a client and Dock 1 a server which offers a video stream. To establish a connection, Dock 2 will need the IP-address of Dock 1. A downside of explicitly using IP-addresses is that they can change over time, resulting in failed attempts to establish a connection between docks. It also requires a system engineer to have knowledge about IP-addresses used within the network to define the topology of an application.

In order for a client to establish a connection, a server must be ready to receive a connection requests, imposing a constraint on the order in which connections are requested. Within the ViFramework a Dock can have both a client and a server role, see Dock 2 (Dock 2 is a client of Dock 1 and a server of Dock 3) in Figure 5.1. If connections are requested when docks are started, depending on the application topology, this can result in a complicated startup
sequence of docks. In cases when an application topology contains a cycle, deployment is impossible, see Figure 5.2, since Dock 1 requires 2 and Dock 2 requires 1. Adhering to this Client-Server connection setup imposes that application topologies can be viewed as directed acyclic graphs (DAG) and does not allow feedback loops.

Another downside of establishing connections in this manner, is that reestablishing connection losses effects other docks. E.g. see Figure 5.1, if the connection between Dock 1 and 2 is lost, Dock 3 needs to be restarted, since Dock 2 has both a server and client role. In the case of large complex distributed applications this could result in an application wide restart, which puts a burden on the system engineer when designing an application topology.

In this chapter we describe a scheme for establishing connections between docks which requires no precedence constraints on connection requests, refrains from explicit IP-addresses and is resilient against failures like connection losses or application crashes.

## 5.2 Dock pairing

The information necessary for establishing a streaming connection between docks are the IP-address of the server and the port on which the server is listening for incoming connection requests. In order to establish a streaming connection between docks this information needs to be communicated. We use OpenDDS to communicate this information. Consequently docks need publishers (P) and subscribers (S) to be able to publish and receive this information, see Figure 5.3. Dock 1 sends its IP-address and a port number to Dock 2. Dock 2 uses this information to setup a streaming connection. Note that when a Dock has both a server and client role it contains both a Publisher and a Subscriber. This way of establishing connections ensures a shift in the proposed Client-Server connection setup, where a server is passive with respect to awaiting connection requests, but now publishes its IP-address and port number.

![Figure 5.2: A cyclic application topology](image_url)

![Figure 5.3: Publisher and subscribers embedded in docks](image_url)
In order for a system engineer to define an application topology publishers and subscribers need to be paired. E.g. the Publisher of Dock 1 needs to be paired with the Subscriber of Dock 2, and the Publisher of Dock 2 needs to be paired with the Subscriber of Dock 3, see Figure 5.3. The connection information published by Dock 1 needs to be distinct from the connection information published by Dock 2, to prevent a connection between Dock 1 and Dock 3.

*OpenDDS* provides two different ways to ensure this distinction, namely by means of topics or domains, see Appendix B.

Figure 5.4 shows an application topology, where different topics (X and Y) are used within a single Domain A.

![Figure 5.4: Publisher and subscribers in different domains](image)

Figure 5.5 shows how an application topology is defined by means of domains. In this example two separate domains A and B are used to ensure distinction. The topics in both domains are the same, Topic X. This does not cause a problem, since the domains are disjoint and a Publisher/Subscriber can only participate in a single Domain.

![Figure 5.5: Publisher and subscribers in different domains](image)

An advantage of pairing docks by means of different domains is that *OpenDDS* facilitates the assignment of an Information Repository to each separate Domain. This redundancy increases the robustness when connections are established.

The ViFramework provides both options of defining an application topology to the system engineer.
5.3 Publisher

A Publisher entity is responsible for communicating connection information, e.g., the IP-address and port number of a Dock to other docks subscribing to this information. Upon initialization the system engineer can specify in which Domain the connection information will be published, see Listing 5.1, note that a Publisher can only participate in one Domain. This allows the system engineer to specify an application topology by means of Domain identifiers or topics.

```
Publisher(domainId, topicType, topicName);
```

*Listing 5.1: Initializing a Publisher component*

5.3.1 Waitsets and conditions

After a Publisher component is initialized, it is ready to send information to any Subscriber in the same Domain. In the publish-subscribe paradigm, publishers are unaware of the presence of subscribers. This gives rise to the problem of how to determine when connection information data needs to be published. One solution could be to periodically publish data samples, see Figure 5.6, were Tp is the publishing period. A downside of this approach is that it induces unnecessary data traffic on the network. Another issue is choosing an appropriate period. If the publishing period is chosen too small the network could saturate, if the period is too large there is a lot of latency in connection setups between docks.

A more sophisticated solution would be to publish connection information when new subscribers enter a Domain and subscribe to the corresponding Topic belonging to the connection information. This requires to detect subscriptions to topics. *OpenDDS* facilitates the detection of (un)subscriptions to a Topic by means of waitsets and conditions, allowing applications to be synchronized with events.

```
StatusCondition condition = datawriter->get_statuscondition();
condition->set_enabled_statuses(DDS::PUBLICATION_MATCHED_STATUS);
WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(condition);
```

*Listing 5.2: Setting a condition on the publication matched event and attaching it to a waitset*

Listing 5.2 shows how to create a condition for the PUBLICATION_MATCHED_STATUS. The PUBLICATION_MATCHED_STATUS is a status that is used as a condition to verify whether there are subscribers registered to the same Topic. Each entity in *OpenDDS* has a
status condition object associated with it. By means of the get_statuscondition() operation an application can access the status condition of that entity. See Listing 5.2 line 1, where the status condition of a data writer is retrieved. The next step is attaching the condition to a waitset. A waitset allows for synchronizing with events that are generated within OpenDDS. See Listing 5.2, where a waitset is created and the PUBLICATION_MATCHED_STATUS status is attached. An application can now be synchronized by means of the wait() statement. The wait operation is blocking until one of the conditions in the waitset evaluates to true, see Listing 5.3.

```c
ws->wait(condition);
```

Listing 5.3: the wait function that blocks until one of the conditions in the waitset is triggered

In this solution the wait operation is blocking until a publication match is detected. After each detection the IP-address and port number are published. This solution is more desirable since it only publishes connection data samples when there is a new subscription to this type of information, keeping the network load at a minimum and reducing latency for setting up a streaming connection.

### 5.3.2 Threading

The publishing of connection information data samples needs to be decoupled from an application, because new subscriptions can be requested during the execution of an application. See Figure 5.7, where a new Dock is added to an application topology. Dock A must be able to handle the new subscription request, without interfering with application tasks running on Dock A.

![Figure 5.7: A new Dock is added to the topology during runtime.](image)

Figure 5.7: A new Dock is added to the topology during runtime.

![Figure 5.8: Threading model for handling subscriptions.](image)

Figure 5.8: Threading model for handling subscriptions.

After a Publisher is created and initialized the function listen() will be started on a separate thread, see Figure 5.8. This function listens for publication matched statuses. Only when
there is a publication match detected, a data sample containing information for setting up a streaming connection will be published. The wait operation suspends the thread until events are detected. It could happen that such a publication match event is missed, if it would be generated at the moment the listen function is not in state 1 \((ws_\rightarrow wait(Pub\text{m\_matched\_stat}))\). However the events generated by OpenDDS are buffered and events generated will be handled correctly.

5.4 Subscriber

A Subscriber is responsible for obtaining data samples containing connection information. The information contained in such a data sample is used to setup a streaming connection. Again the system engineer can specify the Domain and topic name and type belonging to that Subscriber, see Listing 5.4

```java
Subscriber (domainId, topicType, topicName);
```

*Listing 5.4: Initializing a Subscriber*

5.4.1 Listener

The information received by the data reader is made available to an application by means of a listener. Each entity in openDDS defines its own corresponding listener interface. Applications can implement this interface and then attach the listener implementation to that entity. Each listener interface contains an operation for each status that can be reported for that entity. The listener is asynchronously called back with the appropriate operation whenever a qualifying status change occurs. We have implemented our own listener and attached it to the data reader, see Listing 5.5.

```java
connection_info_listener (new ConnectionInfoDataReaderListenerImpl);
dr = this→sub→create_datareader (connection_info_listener);
```

*Listing 5.5: Creating a listener and attaching it to a data reader*

5.4.2 Threading

Figure 5.9 shows that after the Subscriber is created a listen function is started on a separate thread. This function listens continuously for published connection information data samples. If a connection information data sample is received, the IP-address and port number are set. The IP-address and port number are used to setup a streaming connection. If no data sample is available the thread is temporarily suspended, thus the `getData()` function (in Figure 5.9) employs active polling.

If the listen function would be part of the main process and is not a separate process, it could cause problems in certain scenarios. Lets assume that two docks A and B are running and a streaming connection is setup, see Figure 5.10 step (1). If Dock A crashes, the connection between the two docks is lost, see Figure 5.10 step (2). However Dock B is still running. When Dock A is restarted all Domain Participants that are subscribed to the connection information will receive new data samples, see Figure 5.10 step (3).
But since the data reader is only accessed in the initialization of the Subscriber the information is not used. A solution would be to restart Dock B. As explained in the introduction of this chapter this can cause complicated restarts of the distributed application. In order to circumvent this issue, a separate thread is created that continuously listens for all incoming data samples. If the Dock is not connected the information is used to setup a streaming connection and is ignored when a streaming connection is already established, see Figure 5.11.
5.5 Auto discovery

The sequence diagrams in Figure 5.12 and Figure 5.13 provide an overview of how docks A and B are able to establish a connection by means of auto discovery. Note that the difference between the sequence diagrams is the order in which the docks are started. In this sequence diagrams it is assumed that both the Publisher and Subscriber reside in the same Domain and have registered the same Topic.

When a Publisher is created it registers with the information repository. The information repository keeps track of all registered topics and the corresponding publishers and subscribers. When a new Publisher or Subscriber (de)registers for a Topic all publishers and subscribers that are registered to that Topic are notified.

In the sequence diagrams only a single Subscriber is present. If more subscribing docks are present, the Publisher will send connection information to all subscribers. I.e. a Publisher does not discriminate between new and old subscribers. A Subscriber is responsible for handling the connection information.

Figure 5.12: Auto discovery of Dock A and B, where Dock A is started first
Figure 5.13: Auto discovery of Dock A and B, where Dock B is started first
Chapter 6

Network interfaces for streaming

This chapter describes modifications made to the provided proprietary streaming library, *libViStreaming*. It describes the network interfaces for streaming video over the network. These network interfaces are used by Docks for sending and receiving streaming video and associated data. These interfaces implement functions of *libViStreaming*’s client and server for setting up streaming connections and introduce buffers and threads for sending and receiving data and information. The interfaces are called Sender and Receiver.

6.1 Modifications to *libViStreaming*

This section describes the main modification that is made to the proprietary *libViStreaming* library. Other modifications and hands on experiences are described in Appendix C. These modifications are made in the scope of our project to make transparent solutions and to simplify the use of the streaming library for the system engineer.

6.1.1 Setup video input streams

A server dock is able to create video output streams for sending streaming video over the network. Each video output stream at the server side has an identifier and certain codec settings. The identifier is used to identify the stream and the codec settings are used to compress video frames of the video output stream.

For a client dock to be able to receive video frames, video input streams are defined. A video input stream needs to have the same identifier and codec settings (for decompression) as the corresponding video output stream at the server side. This is done manually in the original *libViStreaming*, i.e. the programmer has the responsibility that the identifiers and codec settings at the server and client side correspond with each other.

![Diagram](image)

*Figure 6.1: The client creates video input streams according to the received streaming information from the server.*
In order to simplify the use of the library for the system engineer and to get a transparent solution, the video input streams at the client dock are created automatically according to the video output streams at the server dock. When a client dock connects to a server dock, a list of all the stream identifiers and corresponding codec settings is requested by the client dock. The server dock sends the stream identifiers and codec settings by means of the UD channel and the client dock creates a video input stream for each stream identifier with the associated codec settings, as shown in Figure 6.1.

### 6.2 The interfaces

The network interfaces, called Sender and Receiver, can be used by docks for sending and receiving streaming data and information over the network, like video and meta data.

The Sender is responsible for listening for incoming connection requests from Receiver objects on the same or other hosts. When a connection with a Receiver is established, the Sender is able to send streaming video with its associated meta data, using one or more VideoOutputStreams and OutputStreams respectively.

The Receiver is responsible for requesting and establishing a connection to a Sender. This connection is used for receiving streaming data with its associated metadata from the Sender. On connection the Receiver requests information from the Sender that is needed for setting up input streams. These input streams correspond to the output streams on the server side and are used for receiving streaming video and its associated data using VideoInputStreams and InputStreams respectively.

Next to sending and receiving streaming data, the interfaces are able to send and receive events and data using the EventChannel and UDChannel.

A dock can operate as a server, a client or both depending on the interfaces supported by the dock, see Figure 6.2.

![Figure 6.2: A Dock acts as a client and/or a server. a) As a client, receiving data; b) As a server, sending data; c) As a client and a server at the same time, receiving and sending data.](image)

### 6.3 Connecting Docks

This section describes how two docks establish a connection. One dock sends data and contains a Sender interface and the other dock receives the data and contains a Receiver interface. When the Sender is created it starts listening for incoming connection requests on a specific port. A Receiver is able to request a connection to a Sender when the location (IP-address) of a data producing dock and the port of a Sender on that dock are known, see Figure 6.3a.

The connection between a Sender and a Receiver is established using a handshake protocol. When the handshake is successful the connection between the Sender and Receiver is established, as displayed in Figure 6.3b.
Once connected, the Receiver requests the Sender’s streaming information using the UD channel to set up its input streams (see Figure 6.3c). The streaming information sent by the Sender (Figure 6.3d) consists of information about the video streams and the data streams (used for sending meta data). The streaming information consists of two kinds of data:

- **Video streams**, the identifier and codec settings of the stream;
- **Data streams**, the identifier of the data stream and the identifier of the video stream the data belongs to.

The video stream information is used to setup VideoInputStreams at the Receiver’s side. The data stream information is used to setup InputStreams at the Receiver’s side and attach them to the correct video input stream.

![Figure 6.3: Setting up a connection between two Docks. a) Request a streaming connection; b) Perform the handshake protocol to setup a connection; c) Request the streaming information; d) Send the streaming information.](image)

### 6.4 Buffering

To prevent the situation where sending and receiving of frames is blocking the application, buffers are introduced for each video stream. This way the sending application can put a video frame into the video output buffer and continue without having to wait for the video frame to be sent.

When adding a new video output stream to the Sender, the size of the buffer can be specified, see listing 6.1. The buffer size is set to 5 frames by default. A buffer can use different policies, depending on the use of the application. Currently, only the first-in-first-out (FIFO) policy is supported for the buffers and is therefore used as the default policy. If a buffer is full and a new video frame is received, this video frame is dropped when using the default buffer policy.

```c
1  void addOutputStream(strem_t streamId, CodecSettings codecSettings,
2                     int bufferSize = 5);
```

*Listing 6.1: Adding a video output stream specifying the id; the codec settings and the buffer size (in number of frames)*

The video input streams are added to the Receiver according to the streaming information (identifiers and codec settings) received from the connected Sender. The buffer size of a video input stream is not dependent on the buffer size of the corresponding video output stream at

57
the Sender side. Since the Receiver does not know on creation how many video input streams will be added, the buffer size for each individual video input stream cannot be determined beforehand. Therefore a default buffer size for all video input streams is determined on creation of the Receiver and is the same for all video input stream buffers.

When all input streams are created with the default buffer size, that is set to 5 frames by default, their buffer size should be adapted dynamically. This adaption of the buffer size can be made dependent on e.g. the throughput of the application, since this determines how long a video frame has to wait in the buffer. The throughput can be calculated and the buffer size can be dynamically adapted by the application. Currently the dynamic adaption of the buffer sizes is not supported by the framework but is considered as a recommendation for improving the ViFramework.

Figure 6.4 shows two docks that are connected with each other having two video streams. It gives a visual representation of the buffer sizes on the Sender and Receiver side. The video output streams at the Sender have different buffer sizes. The buffers on the Receiver have the same buffer size and are independent of the buffer sizes of the video output streams.

Figure 6.4: The buffer size on the Sender side can be specified for each output stream independently. The buffers of the input streams on the Receiver side all have the same size.

6.5 Threading

In order for the application to continue running when a frame has been sent, a thread for sending the video frames is created for each buffer, as shown in Figure 6.5. This thread monitors the buffer. When a video frame is available in the buffer the thread sends the video frame to the Receivers that are connected to the Sender. When the buffer is empty the buffer thread is blocked.

Figure 6.5: Creating a video output stream and spawn a thread to read the corresponding buffer and send the frames to all connected clients.
When a video frame is received by a receiving dock the application should not be interrupted. A (listener) thread for receiving video frames is created for each buffer, as shown in Figure 6.6. When a video frame is received by the listener it is put in the corresponding buffer. This task runs on a thread separate from the main application. When the application needs a video frame it can get it from the buffer. When there is no frame available in the buffer the application is blocked until a video frame becomes available.

![Figure 6.6: Creating a video input stream and spawn a thread for receiving video frames.](image)

### 6.6 Sending and synchronizing associated data

Apart from sending video frames, data can be sent using a data stream. When the data is associated to a video stream, the data stream needs to be attached to the video stream.

At the Sender side, a buffer and a thread are created for sending the data to all connected receivers. The data is sent together with the video frame it is associated with.

At the Receiver side the buffer used for the video stream is also used to store the attached data. The buffer for a video input stream contains video frames and associated data. The synchronization of the video stream and its associated data relies on the `libViStreaming` library. The video stream and its associated data are synchronized in the Synchronization layer of the streaming library on reception, before putting them into the buffer. Thus the video input stream buffer contains synchronized samples. Retrieving a sample from the buffer always retrieves the frame along with the associated data. When no data is associated with a video input stream, the place in the buffer used for associated data is empty.

### 6.7 Network interface structure

Now that the notion of buffering, threading and associated data is explained, a detailed structure of the Sender and Receiver interfaces is given, as illustrated in Figure 6.7. A Sender and Receiver always contain an UD and Event channel, for sending and receiving unsynchronized data and events via TCP, as described in Section 6.2.

A Sender can contain one or more VideoOutputStreams for sending streaming video and OutputStreams for sending associated data via RTP. Note that a VideoOutputStream does not need to have associated meta data. A Receiver contains VideoInputStreams for receiving video streams. When data is associated and attached to this VideoInputStream, an InputStream is created and the video data is synchronized with its associated data before it is put in the buffer.
6.8 Multiple senders and receivers

In the proprietary streaming library a client dock receives each video stream that is offered by the server dock, even when these video streams are not used. It is not possible for the client dock to request a subset of video streams offered by the server dock. It is also not possible for a client dock to receive streams from multiple server docks.

To be able to distinguish between streams that are sent to multiple client docks and received from multiple server docks, multiple Sender and/or Receiver objects can be added to a Dock. This however has to be defined at design time.

Adding multiple Sender/Receiver objects can be useful when for example a dock produces multiple streams (e.g. from multiple cameras) and multiple docks are connected that are interested in different video streams. In this case multiple Senders can be used in the server dock, as illustrated in Figure 6.8, to prevent unnecessary transmission of data over the network to all connected clients. When a client dock wants to receive streams from multiple server docks, multiple Receiver objects can be used to distinguish between the streams, as illustrated in Figure 6.9.

---

6.7 The different interfaces within the Sender and the Receiver.

---

Figure 6.8: Dock A contains two Sender components. One sending stream 1 and the other sending stream 2. This way a distinction between different receiver applications can be made, without sending data over the network unnecessary.
Figure 6.9: Dock A contains two Receiver components. This way a distinction between different sending applications can be made.

Since the input streams at the Receiver are created based on the output streams of the Sender, using only one Sender results in all streams being send to each Receiver. When a Dock is not interested in all streams, some streams are sent to the Receiver without being used. These streams are sent unnecessary, consuming network bandwidth.

Using multiple Senders and Receivers for each Dock allows the system engineer to create complex system architectures at design time.
Chapter 7

Dock structure

This chapter describes how the auto-discovery mechanism and network interfaces are combined to form a Dock. An application is described that is used as a proof of concept and that demonstrates how the network interfaces are used.

7.1 Dock decomposition and application interfacing

Figure 7.1 shows the structure of a Dock, containing an AppSender and an AppReceiver component. It is not required for a Dock to contain both. In case a Dock serves only as a source or a sink, only an AppSender or an AppReceiver is necessary.

![Figure 7.1: The structure of a Dock.](image)

The AppComp, being a software component of the application, interfaces directly with the AppSender and AppReceiver components. The ViFramework is designed based on requirements that are specific for usage within ViNotion. A Dock provides optional interfaces that allow the following data types to be used by an AppComp:

- **Frame**, this frame is defined by ViNotion;
- **Event**, an unsigned integer of 32 bits;
- **Data**, which can contain any type of data, but requires conversion to bytes. This conversion should be done by the AppComp.

7.2 Design

A Dock consists of several classes, as shown in Figure 7.2. The Publisher (Subscriber) and Sender (Receiver) classes, as described in Chapter 5 and Chapter 6, are combined to form a
single class, AppSender (AppReceiver). An application can access the auto-discovery mechanism, provided by OpenDDS (Publisher/Subscriber), and network interfaces, provided by *libViStreaming* (Sender/Receiver), by means of the AppSender and AppReceiver classes. This ensures transparent use for a system engineer, since data can be sent and received by means of separate objects.

The SenderConsole and ReceiverConsole classes are used by the Dock to display information about sending and receiving.

The OpenDDS and *libViStreaming* classes are loosely coupled. This enables framework engineers to replace either the auto-discovery mechanism or the streaming software or both, with an alternative.

Figure 7.3 shows how OpenDDS and *libViStreaming* are interfaced with each other. Inside the AppSender, the Publisher and Sender components are decoupled, while the Subscriber
and Receiver components inside the AppReceiver interface with each other. The Receiver component requires an IP-address and port number for establishing connections, which is provided by the Subscriber component. When replacing one of these software components this interfacing needs to be taken into account.

7.3 Proof of concept

This section shows the design and use of an application developed by means of the ViFrame-work as a proof of concept. For the proof of concept a simple application is used that demonstrates all aspects of the framework.

7.3.1 Application topology

The VCA application used for the proof of concept is a pipeline of three stages. Each component of the application is aggregated into a single dock (this process is described in Scenario 2.4.2). The VCA application is constructed of the following three components:

1. The first component reads frames from a camera and sends these frames to the second component;

2. The second component performs a VCA operation, e.g. invert the image colors. This operation is performed on the received frames. The processed frames are sent to the next component along with associated data, that is sent using a separate stream, containing the frame number used for displaying purposes;

3. The third component displays the result and demonstrates the synchronization of video data with associated data.

For the proof of concept two domains are used and the sending and receiving interfaces are all subscribed to the VCA topic, as illustrated in Figure 7.4. Setting the domain, the topic and the topic type is done for each Dock on creation of the sending and receiving interfaces. When a Dock is started, the application is started on a separate thread e.g. read frames from a camera or perform a VCA algorithm. Running the application on a separate thread allows OpenDDS components to detect new docks in the domain, in order to establish connections, while the application is running.
7.3.1.1 Sending video and its associated data

In order to send video and its associated data to one or several receiving docks, the sending dock performs the following set of actions:

- Create a sending object for setting up streaming connections with receiving applications;
- Declare output streams and attach them to the sending object;
- Send the video and associated data using the sending object.

The sending object, called AppSender, is created and initialized when starting a Dock. The creation of an AppSender, as illustrated in Listing 7.1, specifies the domain identifier (domainId), topic type (topicType) and topic name (topicName), which are used by OpenDDS components to create the system topology. The underlying Sender listens for incoming connections from receiving docks on the specified port number (senderPort). DDS components distribute the location (senderIP) and the port number (senderPort) of the sending dock to all receiving docks that request a streaming connection with the sending dock.

```c
AppSender (domainId, topicType, topicName, senderIP, senderPort);
```

Listing 7.1: Initializing a sending object

Once the sending object is created and initialized, the system engineer is able to declare and add output streams to the sending object, as illustrated in Listing 7.2. A video output stream can be added to the sending object, specifying the stream identifier (streamId), the codec settings (codecSettings) and optionally the buffer size (bufferSize) (which is set to 5 frames by default). Next to adding a video output stream for sending video, an output stream for sending associated data can be added to the sending object and can be attached to a video output stream. This is done by specifying the stream identifier (streamId) and the identifier of the stream it is attached to (attachedTo). The buffer size (bufferSize) can also be defined optionally on creation of the output stream. Note that the stream identifiers have to be unique.
Add a video output stream to the sending object.

```c
appSender->addOutputStream(streamId, codecSettings, bufferSize);
```

Add a data stream to the sending object

```c
appSender->addOutputStream(streamId, attachedTo, bufferSize);
```

Listing 7.2: Adding video and data output streams to the sending object

When the sending object and its output streams are created and initialized the sending application is able to send video frames and associated data using the sending object. Listing 7.3 shows the implementation for sending video and associated data. The associated data will be synchronized with the video stream on arrival at the receiving dock. When sending video to docks that are connected, the stream identifier of the stream (streamId) and a video frame (frame) have to be specified. To send associated data with the video stream, the stream identifier (streamId) and the data (data) need to be specified.

Send video frame.

```c
appsender->sendVideo(streamId, frame);
```

Send the associated synchronization data (SD).

```c
appsender->sendSDData(streamId, data);
```

Listing 7.3: Send video frames and associated data called from the application via the sending object

7.3.1.2 Receiving frames

In order to receive video and its associated data from a sending application, the receiving dock performs the following set of actions:

- Create a receiving object for setting up streaming connections with sending applications;
- Get the video and associated data using the receiving object.

The receiving object, called AppReceiver, is created and initialized when starting a Dock. Listing 7.4 shows the code to create an AppReceiver. At the creation of the AppReceiver the domain identifier (domainId), topic type (topicType) and topic name (topicName) have to be specified and are used by DDS components to create the system topology. The buffer size (bufferSize) is optional and specifies the default size of the buffers for the input streams.

```c
AppReceiver(domainId, topicType, topicName, bufferSize);
```

Listing 7.4: Initializing a receiving object

Once the AppReceiver is created and there is a sender object (AppSender) within the same domain (domainId) that is subscribed to the same topic (topicName and topicName) as the receiving object, the receiving dock requests a streaming connection to the sending dock. The receiving dock is able to find the associated sending dock using the location (IP-address) and listening port of the sending dock that is published by means of the DDS components. When setting up the streaming connection, the receiving dock adds input streams to the AppReceiver for receiving video and associated data from the sending application. Finding
an associated sending application, setting up the streaming connection and adding the input streams is done automatically on creation of the AppReceiver.

Once a streaming connection is created between the receiving dock and a sending dock, the receiving application is able to get the received video and associated data from its buffer using the code listed in Listing 7.5. One buffer place (frameData) consists of the video frame as well as the synchronized associated data. When getting video data, the stream identifier (streamId) has to be specified to retrieve the data from the correct stream. Once retrieved, the video frame and the associated data can be extracted from the frame data.

```c++
// Definition of the frame data
typedef pair<image<>, DataBuffer> frameData;

// Get the framedata received by the AppReceiver
frameData fd = appreceiver->getFrame(streamId);
image<> frame = fd.first;
DataBuffer md = fd.second;
```

Listing 7.5: Get frames and corresponding meta data via the AppReceiver

### 7.3.2 Conclusion

This section describes how the proof of concept system is deployed. The docks of the system are deployed on three separate hosts. The Information Repository is deployed on the same host as the dock running the source application. The docks need to know the location of the Information Repository to be able to register, this is done using a configuration file containing the IP-address of the host running the Information Repository.

Before docks can be started, the Information Repository needs to be operational. Next the docks can be started on the different hosts in random order. Once registered with the Information Repository, streaming connections are set up and docks start streaming video and associated data.

While the system is running we have the possibility to terminate a dock and replace it with another dock, that has the same interfaces and output streams but contains a different application, e.g. mirroring an image. Next to replacing a dock, multiple docks can be added to the system at runtime, e.g. adding a sink dock within domain 1067 that subscribes to the VCA topic. This additional sink dock will connect to the VCA dock and receive the frames and associated data. This shows that the system can dynamically be adapted and maintained at runtime without having to restart the whole system.
Chapter 8

Conclusions and recommendations

In order to design a suitable framework that allows distribution of monolithic software developed by ViNotion, a case study is performed. Based on this case study several scenarios from the perspective of a system engineer are created. Every scenario is analyzed that results in a set of requirements. Based on these requirements, a design for the desired framework, called ViFramework, is proposed. A survey is conducted to inventory available frameworks. During this survey all inventoried frameworks are compared and assessed based on the identified requirements. Based on this assessment OpenDDS is selected as a suitable basis for the ViFramework.

The current status of the ViFramework facilitates distribution of a video content application by means of docks. A system engineer can plug software components into a dock. These docks are used to create distributed video content applications and allow the system engineer to specify custom application topologies.

Docks can be deployed on the same or on different hosts in a network and are able to communicate streaming video data with associated synchronized data, events and unsynchronized data over a network.

In order to facilitate inter- and intra dock communication, the ViFramework provides an auto-discovery mechanism. This allows the distributed video content application to automatically resolve the application topology defined by the system engineer. The auto-discovery mechanism is able to cope with run-time topology changes made by the system engineer, like removing, adding or replacing a dock.

The streaming software and the auto-discovery mechanism are decoupled in the ViFramework, allowing either to be easily updated or replaced with an alternative. This allows the ViFramework to be adapted to new emerging requirements and changes in development methods of ViNotion.

The application inside a dock is decoupled from the underlying middleware. This prevents the application to become blocking on the communication of data. The ViFramework offers a simple and transparent interfacing between a dock and an application.

8.1 Scenario overview

An overview of the scenarios, as described in Section 2.4 and Section 2.5, and their support by the ViFramework is given in Table 8.1.
### 8.2 Recommendations

In this section we indicate further directions for research and design suggestions to extend the ViFramework.

#### 8.2.1 Component model

The current implementation of the ViFramework does not support component based software development, but only provides middleware to enable communication between docks. In order
to enable component based software development, conceptual elements for a component based framework are introduced in Chapter 2. Such a component model should not only define the syntax and semantics of components, but also allow the distribution of components. For instance by introducing the concept of a connector. Two different type of connectors between components can be used, e.g. local connectors and network connectors. The streaming middleware we propose can be used as implementation for such network connectors.

8.2.2 Communication

This section describes recommendations related to different aspects of communication.

8.2.2.1 Requesting streams

Currently, the client dock receives each video stream that is offered by the server dock. It is not possible to request a subset of video streams offered by the server dock, unless the system engineer creates multiple sending and/or receiving objects. Functionalities should be added to offer a choice between available video streams from the server dock. This way a client dock can request information about available streams and select the desired video streams. The server dock needs to manage which video streams are sent to client docks.

8.2.2.2 Multicasting

Currently, the server dock sends a video stream to all client docks separately (unicasting), even when these client docks are on the same host. The server dock should be able to identify whether client docks are deployed on the same host. When several client docks on the same host request the same video stream, the server dock should be able to multicast the output stream. This way the server dock does not have to send a video stream to each client dock separately, wasting network bandwidth.

8.2.2.3 Intra-host communication

Currently all docks communicate through the network interfaces, even when docks are on the same host. The ViFramework should distinguish between inter- and intra host communication. When client and server docks are deployed on the same host they should communicate via shared memory instead of network interfaces.

8.2.2.4 Inter-network communication

Currently, only intra-network communication is supported. This means that the ViFramework only operates within a single local area network. Functionality could be added to the ViFramework to enable inter-network communication. Inter-network communication allows the system engineer to deploy an application across several distinct networks.

8.2.3 Quality of service

The network interfaces introduced within the ViFramework currently only support the first-in-first-out (FIFO) policy for streaming video. Different policies should be available to use within the ViFramework to be able to choose a suitable policy depending on the use of the application.
When streaming video, the frames are buffered. Currently, the number of frames that can be buffered is static. The number of frames that can be buffered should be adapted dynamically depending on e.g. the throughput of the application, since this determines how long it takes for a frame to be processed.

8.2.4 Distribution and application management

The ViFramework currently has no support for a full fledged distribution and application manager. The application manager should be able to communicate with docks. This is challenging, since the application manager is a part of the framework and should not be changed when developing distributed applications. A solution would be to define a generic interface at the dock and the application manager that (de)marshals the application parameters to a common format.

8.2.5 Webserver

A webserver is not part of the current implementation of the ViFramework. The webserver allows managing of the entire application from outside a local area network on which it is deployed. It uses the application and distribution managers as services. The option of integrating DDS with a webserver needs to be explored.

8.2.6 Resource management

Currently, the implementation of the ViFramework does not support resource management. A first step towards resource management would be to measure resource consumption with respect to cpu, memory and bandwidth. Based on this information, docks could be manually mapped to available hosts, provided that for such hosts a resource model is available.

The next step would be that the ViFramework facilitates the integration of an algorithm for mapping resource models of components to resource models of underlying hardware. This allows automatic mapping of docks to hosts. Such automatic mapping could also be incorporated in the distribution manager, allowing automatic load balancing of the distributed application. OpenDDS can be used to facilitate the communication of performance data.

8.3 Selecting a suitable framework

Selecting a suitable existing framework requires taking into account the design and development approach of ViNotion. This results in certain pragmatic requirements imposed on the framework that narrow the set of suitable frameworks. If no such pragmatic requirements are present, we recommend to use another framework, namely SOFA 2.0 [8].

SOFA 2.0 contains both the distributed aspect (middleware) and component based software development aspect. The SOFA 2.0 framework facilitates two different types of connectors to bind components, namely design and runtime. Design time connectors are used in the SOFA component model. Design time connector specifications consist of communication styles associated with component interfaces. Runtime connectors are code artifacts that implement design connectors. This approach is much more generic than the solution we propose, since it is not bound to specific interfaces.
Responsibility Matrix

Table 8.2 shows who is responsible for which chapter of the Master Thesis.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Mathijs Opdam</th>
<th>Koen van Langen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: Introduction</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 2: Requirements</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 3: Survey</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 4: Streaming library</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 5: Auto discovery</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Chapter 6: Network interfaces</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 7: Dock structure</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chapter 8: Conclusion</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Appendix A: Frameworks</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Appendix B: DDS</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Appendix C: LibViStreaming</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8.2: Responsibility Matrix.

When we started the project there was no clear partitioning of the assignment. This is why we did the first part of the assignment together. Creating the scenarios, determining the requirements for the framework, researching existing frameworks and investigating OpenDDS and libViStreaming are performed in collaboration with each other.

Once we started to implement concepts of the ViFramework based on OpenDDS and libViStreaming a clear separation emerged. Mathijs Opdam is responsible for the OpenDDS part. I.e. The auto discovery of docks (including robustness against topology changes). Koen van Langen is responsible for the libViStreaming part. I.e. creating the Sender and Receiver classes for setting up streaming connections and send streaming video and synchronizing it with meta data.
Bibliography


Appendix A

Frameworks

Overview

The following frameworks have been considered:

- CANTATA
- Fractal
- GStreamer
- JavaBeans
- Koala
- KobrA
- OpenCCM
- PECOS
- Pin
- Robocop
- SOFA2
- Web Services
- .NET

Framework descriptions

This paragraph gives a brief description of the frameworks. For the full details please refer to included references.
Cantata

The CANTATA (Content Aware Networked Systems Towards Advanced and Tailored Assistance) platform is a set of rules and ways of working to build streaming applications using components with predefined interfaces using common infrastructure (platform). The ultimate technological goal of the CANTATA project is to develop a system that is fully content aware and has understanding of the content that it is processing. Such an ideal system would apply this knowledge to establish an action, or autonomously control the environment where the content is used. For example, a surveillance system would not only detect and identify a criminal act, but while recording it, inform the police, or initiate building controls so that the suspects are captured. This, from an educational point of view, is probably exactly what we want from a framework, but from a business point of view this framework may be too flexible and complex to use with production of real applications. When producing a relative simple product, understanding of the complex framework is required and lots of functionalities may not be required. Besides we are not sure if we are allowed to use and adapt the framework according to our likings/requirements.

Further reading: http://www.hitech-projects.com/euprojects/cantata

Fractal

Fractal is a modular, extensible and programming language agnostic component model that can be used to design, implement, deploy and reconfigure systems and applications, from operating systems to middleware platforms and to graphical user interfaces. The goal of Fractal is to reduce the development, deployment and maintenance costs of software systems in general. A component in Fractal can be viewed as runtime entity that behaves like an object. Fractal uses IDL to define interface which are implemented by components in specific programming language. Each Fractal component is comprised of a content and a controller. The content contains the interfaces and implementation. Invocations can only occur through a component’s interfaces. The controller defines the control behavior with associated component. A parameterized component allows attributes to be changed via its attribute controller interface.

Further reading: http://fractal.ow2.org/specification

Gstreamer

GStreamer is an open source multimedia framework. It is a library for constructing graphs of media-handling components. The applications it supports range from simple Ogg/Vorbis playback, audio/video streaming to complex audio and video processing. GStreamer is released under the LGPL. GStreamer development framework makes it possible to write any type of streaming multimedia application. The GStreamer framework is designed to make it easy to write applications that handle audio or video or both. It is not restricted to audio and video, and can process any kind of data flow. The framework is based on plugins that will provide the various codecs and other functionality. The plugins can be linked and arranged in a pipeline. This pipeline defines the flow of the data. GStreamer can be used on multiple platforms. It has been ported to a wide range of operating systems, processors and compilers.

Further reading: http://gstreamer.freedesktop.org
JavaBeans
JavaBeans technology is the component architecture for the Java 2 Platform, Standard Edition (J2SE). Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications. JavaBeans technology is based on the JavaBeans specification. In JavaBeans a component is a bean, which is represented by a Java class, having methods, event and properties (attributes). Events generated in a bean can trigger methods in other beans by means of event listeners. Thus provide services are modeled by target events and methods in a beans interface and required services by external source events.

Further reading: http://java.sun.com/javase/technologies/desktop/javabeans/docs

Koala
In KOALA, a component is a unit of design which has a specification and an implementation. Components are defined in CDL and an IDL for defining component interfaces. Each component can have multiple interfaces which implements different functionalities. Components can be composed by call through connectors. The KOALA component model distinguishes three kinds of connectors, bindings, glue code and switch. Each new component can be deposited back in the repository.


KobrA
In Kobra, a component is a UML stereotype. Components are constructed as a component specification and deposited into the repository. The repository is a file system which stores a set of UML diagrams. Kobra components are composed by direct method calls into a system template. New composition is not possible during deployment phase since there is no assembler.


OpenCCM
The CCM abstract model offers developers to define interfaces and properties of components. The OMG IDL has been extended to express component interconnections. A component can offer multiple interfaces, each one defining a particular point of view to interact with the component. The four kinds of interfaces are named ports. Two interaction modes are provided: facets for synchronous invocations, and event sinks for asynchronous notifications. Moreover, a component can define its required interfaces, which define how the component interacts with others: receptacles for synchronous invocations, and event sources for asynchronous notifications. The abstract model also defines instance managers – component homes – which are based on two design patterns: factory and finder. A home is defined to create and retrieve a specific component type, and can only manage instances of this type. Nevertheless, it is possible to define several home types for a single component type. The CCM programming model defines the Component Implementation Definition Language (CIDL) which is used to describe the implementation structure of a component and its system requirements: the set
of implementation classes, the abstract persistence state, etc. This language is associated with the Component Implementation Framework (CIF). This framework allows developers to merge the component functional part they have produced and the non-functional part generated from OMG IDL3 and CIDL descriptions. The functional part includes the implementation of the provided interfaces (facets and event sinks). Using the OMG IDL3 definition as well as the CIDL description of the component, a compiler produces the skeleton of the component implementation. This skeleton includes the non-functional part of the component implementation, i.e. un-marshaling GIOP requests, port management, activation, and persistence requirements. These skeletons are implemented on top of APIs provided by containers. Thus, a developer only has to write the functional code in order to complete the component implementation. The compiler also produces the OMG IDL2 mapping as well as a XML descriptor for the component implementation. The OMG IDL2 will be used by component clients. The XML descriptor will be used during the deployment of the component implementation as discussed in the following subsection. The implementation generated part also provides a dynamic introspection API. It includes the operations to discover component ports in a generic manner (same operations for any component type), or in a specific one (operations generated according to the component type). These operations can be used in a dynamic platform to introspect and interconnect component instances at runtime.

Further reading: http://openccm.ow2.org/doc/ccm.html

PECOS

In PECOS, a component is a unit of design which has a specification and an implementation. The inputs and outputs of a component are represented as ports.

Components are composed by linking their ports with connectors. Every component in PECOS has a name, a number of property bundles, a set of ports, and behavior.

Ports are for data exchange, which is the only form of interaction between components with their environment (and, hence, other components). A port is specified with a unique name within a component, the type of the data passed over the port, the range of values that can be passed on this port, and the direction of the port, that is, in, out, and in-out. A port can only be connected to another port having the same type and complementary direction.

The behavior of a component is a function or an algorithm that takes data available on the component ports or some internal data and produces data on the component ports.

In PECOS, there is no component repository. In the design phase, each system or component is specified in the CoCo language in a top-down manner, that is, in terms of compositions of subcomponents.

Pin

Pin is a basic, simple component technology for building embedded, safety- and time-critical software and for demonstrating the feasibility of prediction-enabled component technology. Pin implements the container idiom for software components. Containers provide a prefabricated "shell" in which custom code executes and through which all interactions between custom code and its external environment are mediated. The interface of a Pin component comprises a set of communication channels called pins. Pins support either a synchronous or asynchronous style of communication. Components are strictly reactive, behavior is triggered by the arrival of stimulus on sink pins; response to stimulus is emitted through source
pins. Components are fully encapsulated, the only communication paths from a component to its environment are through its pins. Interaction among two components is enabled by connecting the source pins of one component to the sink pins of another.

Robocop

The aim of Robocop is to define an open, component-based partial architecture for the middleware layer in high-volume embedded appliances that enables robust and reliable operation, upgrading and extension, and component trading. The appliances targeted by Robocop are consumer terminals such as mobile phones, set-top boxes, and network gateways. The architecture, however, aims to be equally applicable for high volume professional devices such as PLCs and certain medical devices. The Robocop architecture defines two interrelated frameworks: the development framework and the execution framework. The development framework defines a number of aspects of the development, trading, and downloading of Robocop Components. The execution framework defines the execution environment for Robocop Components and certain aspects of their dynamic behaviour. The frameworks are linked through the definitions of Robocop Components and the download architecture.


SOFA2

SOFA 2 is a component system employing hierarchically composed components. It is a direct successor of the SOFA component model, which provides the following features: ADL-based design, behavior specification using behavior protocols, automatically generated connectors supporting seamless and transparent distribution of applications, and distributed runtime environment with dynamic update of components.

From its predecessor, SOFA 2 has inherited the core component model, which is however improved and enhanced in the following way: (1) the component model is defined by means of its meta-model; (2) it allows for a dynamic reconfiguration of component architecture and for accessing components under the SOA concepts; (3) via connectors, it supports not only plain method invocation, but in fact any communication style; (4) it introduces aspects to components and uses them to clearly separate the control (non-functional) part of components and to make it extensible. SOFA 2 is not only a tool for modeling components, but it provides a complete framework supporting all the stages of an application lifecycle from development to execution.

Further reading: http://sofa.ow2.org

Web Services

Web Services are fundamental elements of distributed applications in Service-Oriented Computing. A Web service is a piece of binary code designed to support interoperable machine-to-machine interactions for resource sharing over a network. It has an interface described in a machine processable format, specifically in the WSDL (Web Services Description Language). Web services interact with one another via SOAP (Simple Object Access Protocol) messages, typically conveyed by using HTTP with a XML serialization in conjunction with other Web-related standards. The manner in which a Web Service handles SOAP messages is prescribed
by its WSDL interface. WSDL defines the message formats, data types, transport protocols, and transport serialization formats that should be used between services.

Services can be implemented in any programming language and deployed on server machines that are publicly available. Interfaces of services are published in a Universal Description, Discovery, and Integration (UDDI). In the design phase, Web services are composed by delegation of method calls through SOAP messages. For a service to be composed with another service, it first locates the server machine for this service by physically specifying its address in the code so that the two services could send and receive SOAP messages in the design phase.

Services are deployed in the design phase, so there is no separate deployment phase. In the runtime phase, the server of each service provides the runtime environment for the service.

Further reading: http://www.w3.org/TR/ws-arch

.NET

In Microsoft’s .NET, a component is an assembly that is a binary unit supported by CLR. CLR is the runtime environment of .NET that loads, executes, and manages .NET types in the Intermediate Language (IL) into which all .NET languages are compiled. Types of all .NET languages are implemented within the Common Language Specification (CLS), which is a subset of the language features that are adopted by CLR to realize cross-language interoperability. Thus, a .NET component is implemented in any .NET language, including C#, VB, C++, that is constrained by CLS (only features included in CLS can be used) and compiled to IL code, whatever the implementation language is.

A .NET component is made up of metadata and IL code. The metadata of a .NET component contains the following information:

- Description of assembly: Assembly identity, including name, version, and so forth, the files, types, and other resources that make up the assembly, any other assemblies that this assembly depends on, and the set of permissions that are required to run.
- Description of types: Name, visibility, base class, interfaces implemented, and members, including methods, fields, properties, events, and nested types.
- Attributes: Garbage collection; security attributes; version binding, and so forth.

The IL code of a .NET component is the output of a number of compilers of .NET languages (the IL code is actually the CPU-independent instruction set) that is used as the input to a just-in-time compiler in CLR (the IL code is converted to native CPU-specific code by the CLR). Therefore, the metadata is the interface of a .NET component.

.NET components are constructed in Microsoft Visual Studio .NET, which therefore provides a builder for these components. The CLR is the repository and runtime environment of .NET components.

In the design phase, .NET components are composed by method calls through references via metadata. After the design phase comes the runtime phase, with the CLR providing the runtime environment.

Appendix B

Data Distribution Service

This chapter describes the OpenDDS framework, its basic concepts and its architecture.

Introduction

The Data Distribution Service (DDS) is developed in response to the ever growing use of distributed systems. One of the key requirements of distributed systems is to be able to communicate between different processes. Such processes can be distributed across different cores, cpu’s and nodes, where a node can be viewed as a computer or any other electrical system capable of executing processes. When processes are distributed across different nodes, the information is communicated through a transport medium, such as Ethernet or VME Bus, etc. For processes running on the same node, information is communicated via Shared Memory. Figure B.1a, shows a simple distributed application where the Embedded Single Board Computer (SBC) is connected to a temperature sensor. The application on the SBC reads the temperature periodically and communicates the temperature via the network to a workstation. The workstation displays the received temperatures.

To communicate data between different processes (in the example above temperature readings) DDS can be used. DDS uses the data-centric publish-subscribe paradigm. Publish-subscribe applications are typically distributed applications with nodes that communicate with each other by sending (publishing) and receiving (subscribing) data anonymously. An advantage of DDS is that it is implemented as an infrastructure, which allows easy interfacing
with an application, resulting in transparent communication, see Figure B.1b. Note that the application only accesses the publish-subscribe interface and has no knowledge of the underlying layers.

**Data-Centric Publish-Subscribe (DCPS) Paradigm**

In the publish-subscribe model, subscribers only receive a subset of all the messages published. The process of selecting the appropriate messages is known as filtering. There are two common forms of filtering: topic-based and content-based.

1. In a topic-based system, messages are published to "topics". Subscribers which have subscribed to a particular topic will receive instances of that topic, usually called data samples. The publisher is responsible for defining the classes of messages to which subscribers can subscribe;

2. In a content-based system, messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. The subscriber is responsible for classifying the messages.

**General advantage:**

- Loosely-coupled: Publishers are loosely coupled to subscribers and are unaware of their existence. Since the topic is the focus, publishers and subscribers do not require information of the system topology. In the more tightly-coupled client-server paradigm, a client is not able to post messages to a server without setting up a specific connection with that particular server;

**General disadvantages:**

- A downside of the publish-subscribe paradigm is that it can be very difficult to specify properties that the application might need on an end-to-end basis, due to the loosely coupled nature. A publisher is unaware of the presence/absence of possible subscribers. E.g. *When a publisher "assumes" that a subscriber is listening. Suppose that we use a publish/subscribe system that logs temperatures in an environment. An application that reads temperature values and sends the temperature messages to another system that displays them, see B.1a. If the displaying system happens to crash, the publisher won’t have any way to see this, and all the temperature messages will be lost.*

- Publish/subscribe systems tend to scale well for small amounts of publisher/subscribers, but it does not scale well with large amount of participants. If publishers and subscribers are present in large numbers and communicate by means of broadcasts, it can saturate the entire network (so called IP broadcast storms). This effect has a direct impact on the applications communicating by means of the publish/subscribe paradigm. Applications would suffer from highly fluctuating throughput.

- For Publish/subscribe systems that use brokers (servers), it could be the case that brokers send information to the wrong subscriber, since control data that is used to selecting the appropriate subscriber is in-band (control data is communicated over the same channel as the topic data).
DDS falls in the first category and is topic-based filtering. The general disadvantages of the publish-subscribe paradigm mentioned above are addressed in DDS to some extent. The communication used by DDS is also Data-Centric. This allows for the ability to specify certain properties of the data being communicated, like publication- and subscription rate, or data expiration, etc. These Quality of Service policies can be set for each topic.

**DDS Architecture**

The specification of DDS [9] consists of two distinct parts, see Figure B.5, the DCPS (Data-Centric Publish-Subscribe) part and the DLRL (Data Local Reconstruction Layer) part. DCPS is the lower layer that the application can use to communicate with other applications distributed across a network. DLRL is an extra abstraction layer to add more transparency. This layer enables DCPS data fields access through its own object oriented programming classes. Figure B.2b shows how an application is layered for using both DCPS and DLRL. In OpenDDS [15] only the DCPS part is implemented.

![Layered views of DCPS and DLRL](https://via.placeholder.com/150)

**Figure B.2:** Layered views of DCPS and DLRL, taken from *OpenDDS* [15]

Since we recommend using *OpenDDS* [15] to serve as a basis for our framework, we will focus on the DCPS part. For the DLRL we refer to the DDS specification [9].

**DCPS**

The DCPS part distinguishes the following concepts/entities:

- Domain;
- Domain Participant;
- Topic;
- Data Writer;
- Publisher;
- Data Reader;
- Subscriber;
Figure B.3 shows how the entities relate to each other. A data sample is sent and received to/from the domain. Publishers and subscribers manage data writers and data readers. Note that a publisher/subscriber can have ownership over several data writers/readers. Data writers/readers are associated to each other by means of topics. Figure B.3 shows three different types of data streams (1, 2 and 3). Subscribers which have subscribed to topic 1 will not receive any information about topics 2 and 3. This allows for a nice separation of data flows within a domain. In the next section we will discuss these entities/concepts in more detail.

**DDS entities**

In this section we provide a more detailed overview of the components/concepts used in OpenDDS.

**Domains and domain participants**

A domain is used to bind applications together to enable communication. A distributed application can use single domain or multiple domains, see Figure B.4. Figure B.4a shows an example system comprised of 5 nodes, all participating in the same domain.

Figure B.4b shows the use of multiple domains. The use of multiple domains provides an effective way of data isolation. For instance in domain A could be used to communicate temperature reading, while domain B could be used to communicate control data concerning these temperature readings, like the sampling frequency. You could argue that this also could be realized by using different topic types. However using different domains nicely decouples the temperature data from the control data on a functional level, since different Data Writers and Data Readers will be used. The usage of domains also facilitates extensibility, since new functionalities can be added with minimal impact on existing functionalities. An application uses an object called "Domain Participant" to gain access to a particular domain.
Connections between publishers and subscribers are created based on topics. The topic of a publisher must match the topic of a subscriber for communication to take place. If they differ, communication will not take place. In DDS a topic is comprised of a topic name and a topic type. The topic name uniquely defines the topic within a domain and the topic type defines the data being communicated. Topics with different name can have the same topic type. Topic types can be specified by means of the Interface Definition Language (IDL).

The example in Listing B.1 shows a possible data type of messages communicated in Figure B.1a. Note that the sensorId (sensor identifier) is made key in this type definition. When defining a topic type one, or more fields can be made key. The DDS middleware uses key attributes to sort incoming data. Keys are used by DDS to define different instances of the same topic. This is typically handyconvenient when scaling of the application is involved. Suppose that the example explained in Figure B.1a would be scaled up to 4 temperature sensors, which are placed at different positions in an environment. The usage of keys prevents creation of new topics. If we were to add a new temperature sensor, the publishing application fills in the sensorId to differentiate between the values of the other sensor readings. The subscriber will sort the incoming sensor readings and provides it to the application relative to its key value.

**Data Writer and publisher**

Data writers are access points for an application to publish data samples into a domain. Each Data Writer can specify QoS policies. For an extensive overview of QoS policies, we refer to Farabaugh et al [17]. Once a Data Writer is created and the appropriate QoS policies are set a simple write call will publish the data sample into the domain, as shown in Listing B.2.
When the write command is executed the data writer queries the key values (see the subsection on topic) to determine the instance of the data sample. The instance can also be explicitly registered before publishing. Such a registration returns an instance handle to that data sample. Calling the write action with such a handle saves some overhead. When the write operation is executed, the data is moved to the publisher. A publisher can be seen as a container object for managing data writers. Figure B.5a shows the publication model.

![Publication model](a) Publication model ![Subscription model](b) Subscription model

*Figure B.5: DDS publication and subscription model, taken from Farabaugh et al [17]*

### Data Reader and subscriber

A data reader is the entry point for an application to access data samples which are received by the Subscriber. A Subscriber is a container object for hosting Data Readers, allowing to set default QoS policies and event handling routines. Figure B.5b shows how entities are associated with subscriptions. Upon reception the application can be notified that a new data sample is available in three ways:

- **Listener Callback Routine**, a callback routine will be called as soon as a data sample is received by the subscriber. The application designer/programmer can implement actions that need to be executed;

- **Polling the Data Reader**, periodically checking if new data is available;

- **Conditions and WaitSets**, an application can use a WaitSet for synchronisation. A Waitset is a set of conditions in which one of the conditions has to evaluate to true in order for the application to be unblocked. As a safeguard, a timer can be specified to avoid infinite blocking. A

Accessing data can be done either by a read or a take, see Listing B.3. The **take()** command removes the data sample from the buffers in the middleware and the **read()** command allows data samples to be retrieved multiple times.
OpenDDS specifics

OpenDDS is an implementation of the DDS specification [9]. In this section we introduce additional concepts used by OpenDDS.

Information Repository

The Information Repository (IR) is not part of the DDS specification [9] but is a concept used by OpenDDS [15]. A single, separate DCPS Information Repository (DCPSInfoRepo) process acts as a central manager, associating publishers and subscribers.

Auto discovery

Before communication is possible, topics first need to be registered with the Information Repository (IR). New domain participants can retrieve the available topic in a domain and subscribe to it. When a publisher registers a topic with the Information Repository, the Information Repository looks for matching topics which were previously registered. If any matches are found, the participants subscribed to this topic are notified, by sending the associations. Figure B.6 shows this auto-discovery when a publisher registers with the Information Repository.

Mapping to requirements

Composition

OpenDDS does not facilitate component based software development as defined within the context of the ViFrameWork. It is possible to specify docks (application binaries) and their interfaces that allow inter-dock communication and set and get dock parameters.

Distribution manager

Considering the requirements of the distribution manager defined in Chapter 2, OpenDDS does not facilitate a similar concept. The information repository can serve as basis for the design and implementation of the distribution manager. There is no functionality present regarding, a repository for storing docks, nor is it possible to retrieve performance metrics or available resources. These requirements can be implemented using OpenDDS.

Application and dock manager

In OpenDDS no entities like an application manager or dock manager are present. The application manager and dock manager can be developed by means of OpenDDS, since both entities communicate control data for which the publish-subscribe paradigm is suitable.
Streaming facilities

No explicit streaming facilities are present in OpenDDS, streaming can be realized by putting image data in data samples.
Appendix C

Streaming library

ViNotion provided us with an existing streaming library, libViStreaming. This library was built under the GCC compiler and we had to port it such that it also builds under the VCC compiler. While doing this we also found some bugs in the API. We also made some modifications to the library to take away some known limitations, as described in the documentation of the library.

Building libViStreaming

To be able to integrate the libViStreaming library into the OpenDDS framework, we need to have a better understanding of the libViStreaming library. The OOTI students built a demo application to demonstrate the operation of the library. We used this demo application as a starting point to get a better understanding of the library. To do this we first had to port the demo application from CodeBlocks to Visual Studio.

To be able to run the demo application in Visual Studio, libViStreaming has to be build with the VCC compiler. This way the API is supported on multiple platforms and can be used with Visual Studio and produces a DLL library. To build the API with the VCC compiler a new project is created within Visual Studio which imported all the available C++ source files. We had to make sure that all dependencies were correct and all correct libraries (.lib-files) were imported.

To be able to call functions from the classes in the library, the classes needed to be exported to the lib-file. This way the functions can be called from the demo application. The original library uses a wxCanvas object to display the output. We use the VideoOutput object from the libViNotion library to display the output instead.

According to the documentation of the OOTI project, any libViStreaming application has to be a windowed application. The API uses sockets and timers from wxWidgets. A widget window application has to be created to handle socket events.

Bugs

After porting, running and analyzing the demo application under the VCC compiler, we found some bugs in the libViStreaming API:

- When setting up the streaming connection, the destination address contained the address of the server, while this should be the address of the client. Since when filling in
the server address only a (local) connection to the server itself can be established. We changed the destination address to the address of the client:

```c++
// in ServerManager.cpp
outputStream->setDestAddress(clientIPAddress);
```

Listing C.1: Setting the destination address of the output stream

- Sending streams over the network did not work. This was because the default port numbers on the client and server side did not match. The port number the client used for the handshake with the server was 9050, while the port number the server tried to connect to was 9500. This looks like a typo and we changed the client handshake port to 9500:

```c++
// in ClientManager.hpp
bool start(int clientHandshakePort = 9500,
            int serverHandshakePort = 9000);
```

```c++
// in ClientSession.hpp
virtual bool connect(int clientHandshakePort = 9500,
                     int serverHandshakePort = 9000);
```

Listing C.2: Setting the handshake ports

- With the current settings is was not possible to connect multiple clients with one server, because for each session with a client the server wanted to connect through the same port with these multiple clients. This is not possible since the port is in use. We solved this problem by keeping track of the number of clients in the server session and make the port number for connecting to a client dependent on the number of the client.

```c++
// in ServerManager.cpp
int serverStreamPort = m_serverPortBase + 6 + (100 * (sSession->m_serverPortOffsetFactor - 1));
```

Listing C.3: calculating the server stream port

The variable `m_serverPortOffsetFactor` is initialized when the server session is created and contains the number of the client, which is incremented every time a client connects to the server. We took a range of 100 port numbers for each client, so that each client has a sufficient port range and the chance that clients connect to the same port is minimized.

With these bugs solved we are able to send multiple streams over the network to one or more clients. Also Events and Unsynchronized Data can be sent.

Modifications

Multiple clients on the same host

When a connection is set up between a client and a server, a session is created with a session id at the client side. This session id must be unique for each session and is created based on
the IP-address of the client. This means that multiple clients on the same host are unable to connect with the same server, since they run on the same IP-address and thus produce the same session id.

To make sure that every client (session) gets a unique identifier, the identifier must not be explicitly based on the IP-address of the client. To solve the problem with multiple clients on the same host the identifier is based on the IP-address as well as the base port number at which the client is connecting with the server. The system engineer only has to make sure that the clients running on the same host connect through different port numbers.