Instrumentation of networked video streaming components

Bram Kersten, BSc

May 22, 2011

Master Thesis
Eindhoven University of Technology
Department of Mathematics and Computer Science
Security and Embedded Networked Systems

In cooperation with:
ViNotion B.V.

Supervisor:
Dr. Rudolf H. Mak
Eindhoven University of Technology
r.h.mak@tue.nl

Internship Supervisor:
Kris van Rens
ViNotion B.V.
kris.van.rens@vinotion.nl
Abstract

Real-time video content analysis applications for surveillance become more and more demanding. The need for load distribution, remote management and reusability calls for a component framework specialized in networked video streaming applications. Whereas lots of component frameworks exist nowadays, frameworks targeted at networked video streaming are scarce. Added requirements, imposed by video surveillance applications, include real-time computing and quick failover.

In this thesis a framework is proposed that meets these demands by enabling the distributed execution of video streaming applications in an efficient and resource-aware fashion. The design of the proposed framework is presented and a prototype implementation is evaluated. The results of this evaluation show that this implementation is efficient and can successfully perform failover handling, making it suitable for distributing surveillance applications.

The proposed framework is flexible in order to enable fast prototyping and to cope with frequently changing components. Therefore, it is also suitable for usage in a research or developmental phase. Instrumentation of existing components, extending them with platform specifics and network functionality, is generalized. Besides, a number of suggestions towards atomization of this instrumentation process are proposed.
Acknowledgements

This thesis is written as part of my master project that I have performed at ViNotion B.V. in cooperation with the department of System Architecture and Networking at Eindhoven University of Technology. Finishing the master project concludes my study Embedded Systems at Eindhoven University of Technology.

I enjoyed working on this project because of its pragmatic nature combined with the challenge of designing and implementing a framework that facilitates the component-based development of state of the art surveillance software, such as developed by ViNotion. Therefore, I express my gratitude to the employees of ViNotion for creating a pleasant working environment and treating me as “one of the team”. Special thanks goes to my internship supervisor Kris van Rens for his magnificent support along the route. Furthermore, I thank Egbert Jaspers for initiating this project and for his personal involvement.

Secondly, I would like to thank Rudolf Mak very much for all the time put in supervising my work, providing me with great advice and giving me numerous new insights. I am grateful to Rudolf Mak and Kris van Rens for co-authoring the research paper that emerged from this graduation project.

Whereas a student life is not complete without room for social activities, I would like to thank my friends at student association 'Demos' and in particular my friends at fraternity 'Drago' and year association ‘XiniX’ for providing these kind of activities, unforgettable moments and a great environment for social development.

I am grateful for the encouragement that my girlfriend, Soe Kreek, has given me during my graduation period. Last, but definitely not least, I would like to thank my parents for their continuous support and patience throughout my study and for making it possible to complete my Embedded Systems master.


Bram Kersten
Contents

Acknowledgements v
Table of Contents ix
List of Figures xii
List of Tables xiii
Acronyms xv

1 Introduction 1
  1.1 Context and background 1
  1.2 Problem description 3
  1.3 Project goals 3
  1.4 Approach 4
  1.5 Outline 5

2 ViFramework 6
  2.1 Introduction 6
  2.2 Implementing the ViFramework 9
  2.3 Scenarios 10
    2.3.1 PTZ camera scenario 11
    2.3.2 Failover scenario 13
    2.3.3 Expansion scenario 15
    2.3.4 Internet scenario 17
  2.4 Non-functional requirements 18
  2.5 Requirements overview 19

3 Framework Architecture 22
  3.1 Component definition 22
  3.2 Framework contents 25

4 Framework Design 28
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Evaluation</td>
<td>74</td>
</tr>
<tr>
<td>8.1</td>
<td>Requirement fulfillment</td>
<td>74</td>
</tr>
<tr>
<td>8.2</td>
<td>Performance evaluation</td>
<td>75</td>
</tr>
<tr>
<td>8.3</td>
<td>Evaluation method</td>
<td>75</td>
</tr>
<tr>
<td>8.4</td>
<td>Framework overhead</td>
<td>75</td>
</tr>
<tr>
<td>8.5</td>
<td>Failover automation</td>
<td>76</td>
</tr>
<tr>
<td>9</td>
<td>Instrumentation Procedure Generalization</td>
<td>77</td>
</tr>
<tr>
<td>9.1</td>
<td>Dock abstract base class</td>
<td>77</td>
</tr>
<tr>
<td>9.2</td>
<td>Interfaces</td>
<td>78</td>
</tr>
<tr>
<td>9.3</td>
<td>XML description</td>
<td>78</td>
</tr>
<tr>
<td>9.4</td>
<td>Assembly</td>
<td>78</td>
</tr>
<tr>
<td>9.5</td>
<td>Towards automation</td>
<td>79</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Dock descriptions</td>
<td>79</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Code generation</td>
<td>79</td>
</tr>
<tr>
<td>10</td>
<td>Conclusions</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Future Work</td>
<td>83</td>
</tr>
<tr>
<td>11.1</td>
<td>Implementation improvements/extensions</td>
<td>83</td>
</tr>
<tr>
<td>11.2</td>
<td>Research opportunities</td>
<td>84</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>A</td>
<td>Research paper</td>
<td>90</td>
</tr>
<tr>
<td>B</td>
<td>Dock Manager XML Schema</td>
<td>91</td>
</tr>
<tr>
<td>C</td>
<td>Example Dock Manager XML Description</td>
<td>95</td>
</tr>
<tr>
<td>D</td>
<td>Example Dock Implementation</td>
<td>99</td>
</tr>
<tr>
<td>E</td>
<td>Matching algorithm</td>
<td>103</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Dataflow diagram of the virtual fencing application ................................ 7
2.2 Dataflow diagram of the instrumented virtual fencing application .......... 7
2.3 Dataflow diagram of the distributed virtual fencing application .......... 7
2.4 The ViFramework ..................................................................................... 8
2.5 PTZ camera application ........................................................................ 11
2.6 Distributed PTZ camera application (Deployment) ................................. 12
2.7 Failover scenario .................................................................................... 14
2.8 Disconnected failover scenario ............................................................... 14
2.9 Expansion scenario, 2 camera application, expandable to 3 cameras ...... 15
2.10 Expansion scenario, 4 camera application ........................................... 16
2.11 Expansion scenario, load balanced 4 camera application ..................... 16
2.12 Internet scenario .................................................................................. 18
3.1 Idealized life cycle: Category 3 ............................................................... 24
3.2 Deployment diagram ViFramework ......................................................... 26
3.3 High-level class diagram of the ViFramework ......................................... 27
4.1 Streaming component wrapped in a dock .............................................. 30
4.2 Dock Class ............................................................................................. 32
4.3 Dock: State diagram .............................................................................. 34
4.4 DockManager class .............................................................................. 37
4.5 Matrix: resource usage abstraction [1] ................................................... 40
4.6 Left: Host’s CPU usage registered by the dock manager — Right: Resource graph received from the dock manager at distribution manager .............................. 40
4.7 DistributionManager class ..................................................................... 42
4.8 Sequence diagram: Instantiate dock of a given type ............................. 45
4.9 Sequence diagram: Destroy dock ........................................................... 45
4.10 Sequence diagram: Add connection ..................................................... 46
4.11 Sequence diagram: Remove connection .............................................. 46
4.12 Sequence diagram: Start dock .............................................................. 47
4.13 Sequence diagram: Stop dock .............................................................. 47
5.1 Orbweb network structure [2] ................................................................. 53
6.1 Interface matching example: matching video stream interfaces . . . . . . . 60
6.2 Interface matching example: non-matching video stream interfaces . . . . 61
6.3 QoS propagation example . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.4 Object data interface definitions. . . . . . . . . . . . . . . . . . . . . . . . 63
6.5 Interface matching example: interface binding . . . . . . . . . . . . . . . . 64
6.6 Possible network structure supported by the ViFramework. End-to-end
RTP sessions are used for intra-LAN video streaming. For inter-LAN video
streaming the XMPP server is used as SOCKS 5 proxy. Only the XMPP
server requires a public IP address. Metadata communication is not de-
picted in this figure] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.7 Ping-pong buffer. a) Initially a frame can be written to an empty buffer.
On completion, the read and write buffers pointers are switched. b) After
a successful switch, the stored frame can be read and a new frame can be
written concurrently. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.8 SOCKS 5 connection setup data flow [3] . . . . . . . . . . . . . . . . . . 69

7.1 GUI: Host information. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
7.2 GUI: Deployed docks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
7.3 GUI: Topology image. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
7.4 GUI: Log screen. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
7.5 GUI: Topology image of the PTZ scenario. . . . . . . . . . . . . . . . . . 73
7.6 GUI: Connections in the PTZ scenario. . . . . . . . . . . . . . . . . . . . 73
List of Tables

2.1 General requirements ........................................ 19
2.2 Dock requirements ........................................... 20
2.3 Dock manager requirements ................................. 20
2.4 Distribution manager requirements ....................... 21
2.5 Communication requirements ............................... 21

5.1 Network substrate comparison ............................... 54

8.1 Overhead measurements ..................................... 76
8.2 Overtake measurements ..................................... 76

10.1 Unmet requirements ......................................... 82
10.2 Implementation metrics .................................... 82
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBSE</td>
<td>Component-Based Software Engineering</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MUC</td>
<td>Multi-User Chat</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>PTZ</td>
<td>Pan Tilt Zoom</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real-time Streaming Protocol</td>
</tr>
<tr>
<td>SAN</td>
<td>System Architecture and Networking</td>
</tr>
<tr>
<td>SASL</td>
<td>Simple Authentication and Security Layer</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TU/e</td>
<td>Eindhoven University of Technology</td>
</tr>
</tbody>
</table>
UDP  User Datagram Protocol
VCA  Video Content Analysis
ViCoMo  Video Content Modeling
WAN  Wide Area Network
XML  eXtensible Markup Language
XMPP  eXtensible Messaging and Presence Protocol
Chapter 1

Introduction

1.1 Context and background

Over the years video streaming applications have become more and more important in areas such as surveillance and consumer electronics. Due to the huge increase of network capacity it has become feasible to distribute video over a network enabling the development of applications like Video-on-Demand and remote camera monitoring. Applications processing video streams are typically computationally intensive but have very little data reuse. This makes them particularly suitable for the pipes and filters \[4\] design pattern, dividing the task of the application into several sequential processing steps. These steps are connected by the data flow through the system. Using multiple cores makes it possible to parallelize this process by means of pipelining.

Component-Based Software Engineering (CBSE) \[5\] has become an established approach in many application domains. Instead of creating each application from scratch, applications are built using reusable components. The main objective of this approach is to maximize reusability and reduce time to market in order to minimize costs. Furthermore, it enables third-party development of components, enabling larger and more complex applications.

The high resource requirements of video processing algorithms are likely to exceed the available resources on a single host. This results in the need to distribute an application over multiple hosts. Besides that, the need for geographical distribution of applications might arise. Using the component-based approach eases this distribution, because components can be used as a unit of distribution. Assuming multiple hosts are available and connected by a network, load distribution could be performed to meet the application’s requirements. A downside of this method is that data between components, that was communicated locally, now needs to traverse a network. Evidently, this increasing the use of network bandwidth.
Chapter 1. Introduction

This master’s project has been performed in the context of the Video Content Modeling (ViCoMo) project. The ViCoMo project is concerned with the processing and interpretation of video data acquired by multiple cameras. The ultimate goal of the project is to create intelligent and context aware visual systems. By using multiple cameras and computation intensive algorithms to acquire context from video streams, the need for networked video streaming components is evident.

One of the partners of the ViCoMo project is ViNotion, a company specialized in Video Content Analysis (VCA) for surveillance purposes. The applications developed by ViNotion are provided as monolithic systems analyzing a limited number of video streams. The goal of the analysis is to detect and track objects that are visible in the provided video streams. Most of these applications are computationally intensive and need to run in real-time. This combined with the current lack of remote management makes implementation in the form of networked video streaming components very attractive.

An other partner of the ViCoMo project is the Eindhoven University of Technology (TU/e). The System Architecture and Networking (SAN) department of this university contributes to the ViCoMo project by facilitating the software infrastructure that allows runtime composition of video streaming applications. To accomplish this, a combination of Service-Oriented Architecture (SOA) and component-based design is used.

Nowadays, many component frameworks exist, but component frameworks supporting video streaming are rare, especially when network functionality is required. A framework that does support this is required. In order to cope with frequent changes to component code, which is typical for VCA software [7], and with the need for fast prototyping [8], a flexible framework must be developed.

In this master project a framework is presented that enables video streaming components to be developed and deployed on several hosts connected by a network. The framework provides an interface that enables runtime (re-)configuration and composition of distributed components. The framework provides location transparency, enabling components to be developed without requiring knowledge of the physical location the component will be used. It must provide means to connect components independently of their relative locations and must, therefore, provide network functionality.

Because the framework must support remote monitoring and management, network connection must be able to traverse NAT-routers and firewalls. The designed framework does not concern component developers with how the components are connected, what compression method is used and how timing and threading are controlled.

This work has also been published in the form of a research paper submitted to the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications [9] and is included in Appendix A.
1.2 Problem description

Before VCA algorithms such as those developed by ViNotion can be used as components in the manner sketched above, they have to be supplemented with software that provides additional functionality for network usage and that caters for platform specific implementation details. This process is called instrumentation.

In this context the instrumentation of a component includes:

- The definition of which component interfaces need to be made accessible to the environment.
- Identifying Quality of Service (QoS) properties of these interfaces.
- Adding control interfaces that enable remote management of the component.
- Coupling component interfaces to framework interfaces.

A component developer for the proposed framework should be concerned as less as possible with these tasks. Therefore, a generalization of the instrumentation procedure is required. This generalization could facilitate future automation of component instrumentation.

In a previous master assignment carried out at ViNotion by K.S.W van Langen and M. Opdam [10], distribution of a small surveillance application has been investigated. In that project, a framework based on OpenDDS [11] is proposed to arrange a publisher/subscriber kind of communication. They concluded that OpenDDS was suitable for providing network services, but lacked video streaming capabilities which had to be implemented manually. The framework, named ViFramework, has only been partially implemented and no application has been run on it.

1.3 Project goals

For this project three main goals are defined:

1. Implement an existing ViNotion application using the ViFramework on multiple hosts. To that end, further implement the ViFramework and develop proper tooling to easily manage connection between components, relaxing the effort for manually coding/establishing a network of components and enabling (third-party) runtime composition of components.

2. Generalize the instrumentation procedure while keeping in mind that the underlying platform might change in the future. Place special emphasis on video streaming
components.

3. Examine resource requirements for these components, particularly those related to
the communication between components. Propose handling decisions based on the
resource requirements of the components and the available resources of the system.

The framework created in this project is also evaluated in order to prove its usability and
efficiency.

1.4 Approach

In order to meet the goals stated in Section 1.3 an existing ViNotion application will be split
up into components that will run on different hosts. For this purpose, an implementation
of the ViFramework proposed by van Langen and Opdam [10] will be developed and the
components of the application provided by ViNotion will be distributed over multiple hosts
using this framework. Using the knowledge gathered from instrumenting several compo-
nents, guidelines will be developed, preferably to automate the process or achieve machine
support. When the existing application successfully runs on the developed framework,
resource information will be extracted from the different hosts and used for scheduling
purposes. In order to evaluate the implemented framework, the application ported on it
will be tested for correctness and robustness and performance metrics will be calculated.

In the implementation process of the ViFramework five major steps are distinguished:

1. Instrumentation of selected components from existing ViNotion applications, extend-
ing them with network capabilities. Components should be able to register their inter-
faces, at runtime, at a central ViFramework service which will also be implemented.

2. Interface matching of various components by the central ViFramework service in order
to setup connection between them. The creation of point-to-point video streaming
connections between two components has priority here.

3. Generalization of the instrumentation process, listing possible interfaces and try to
distinguish between generic and application-specific interfaces. Finally, creating a
flexible communication protocol.

4. Resource monitoring at the host-level, forwarding this information to the central
ViFramework service.

5. Using the available resource information, the network topology, the defined interfaces,
the preferences of the user and the central service will calculate a good network layout
and load distribution.
1.5 Outline

The first four steps are the basis of the ViFramework whereas the last step is a proof of concept that the framework is capable of performing more complicated tasks (e.g. scheduling). Priority lies with the first four steps.

1.5 Outline

Chapter 2 is dedicated to defining the requirements of the ViFramework using four example application scenarios. Chapter 3 outlines the framework’s high-level architecture design, whereas in Chapter 4 the in-depth design of the framework is presented. Chapter 5 elaborates on the choice of the network substrate the framework is based on. In Chapter 6 the interface design is discussed. The implementation of the ViFramework is presented in Chapter 7 and evaluated in Chapter 8. A best practice and suggestions towards automation of the instrumentation procedure are given in Chapter 9. Chapter 10 draws conclusions from the presented work in this thesis and Chapter 11 gives directions for future work.
Chapter 2

ViFramework

This chapter introduces the ViFramework designed by van Langen and Opdam [10]. It elaborates on which parts of the original framework will be used by the framework proposed in this thesis. Requirements of the proposed framework are gathered by defining four application scenarios.

2.1 Introduction

The ViFramework is explained by means of a small video surveillance application. In this application a camera monitors an area of interest. The area of interest contains a so-called “forbidden” zone. The border between the forbidden zone and the remainder of the area is called a virtual fence. Virtual because this border does not need to be visible to the occupants of the area but may be hidden to them and only be known by the surveillance system. The data-flow diagram of this Virtual Fencing Application is depicted in Figure 2.1 and consists of four components:

1. Capture device: A camera covering the area of interest.
2. Detection Algorithm: Detects persons inside the area of interest.
3. Tracking Algorithm: Determines the path of the detected persons over time.
4. Event manager: Sends an alarm when a person enters a predefined zone.

For use in the ViFramework these components need to be instrumented, extending them with network functionality and platform specifics. This results in the creation of Docks. A dock can be viewed as a unit of distribution containing one or more components. A
2.1. Introduction

possible result of the instrumentation procedure of these four components is depicted in Figure 2.2.

The created docks can be distributed over available hosts using the ViFramework. The framework facilitates the network connections between the hosts. An example distribution of the instrumented virtual fencing application is depicted in Figure 2.3.

Figure 2.4 (taken from [10]) illustrates how the ViFramework is used to deploy and execute the virtual fencing application on a hardware platform consisting of three hosts. The ViFramework consists of the following entities:

- **Software component**: A software component can be seen as a modular software element providing some service to its “clients”. Whereas the inside of the component may be unknown the service it provides and its interface must be well-defined.

- **Dock**: A dock can be viewed as a unit of distribution containing one or more components. Components are placed into docks which provides them with network functionality, means for intra-dock communication and platform specifics. This is the so-called instrumentation process. The components themselves do not need to know anything about the network, this is all taken care of by the docks.
Figure 2.4: The ViFramework
2.2. Implementing the ViFramework

- **Dock Manager**: The dock manager handles installing, removing, starting and stopping of all docks located on the same host as the dock manager itself. The dock manager sets source and destination addresses of dock interfaces and also sets component parameters.

- **Host**: A host is a physical machine in the network running its own dock manager and hosting zero or more docks.

- **Dock Repository**: A database of all available dock types and their capabilities.

- **Distribution Manager**: The distribution manager distributes docks over the available hosts in the network. It can create or break connections between docks and is able to add or remove docks from the dock repository. The distribution manager is capable of automatically distributing and starting applications, but a user interface is also provided.

- **Application Manager**: The application manager handles retrieving, updating and setting application parameters. This can be specified via a user interface or script.

- **Web-server**: Used as an interface for remote application management.

A system engineer can use the web-server as an interface to the distribution manager and the application manager. The web-server provides the system engineer with means to set up, configure and start distributed application.

### 2.2 Implementing the ViFramework

At the start of this master project, the dock manager, application manager, dock repository and the web-server were not implemented. Furthermore, the distribution manager did not support setting up connections between docks. Connections between docks in a network had been addressed, but these connections had to be set up manually by hard-coding the bindings in the dock code.

In order to meet the goals defined in Section [13] the distribution manager will be extended with functionality to remotely deploy docks and to setup connections.

The ViFramework describes an application manager that retrieves, updates and sets application specific component parameters. These parameters are most likely to be set during the distribution process, which is why it makes sense to merge the functionality of the application manager and the distribution manager.

The system engineer will be able to connect docks defined at hosts if they have compatible
interfaces, creating new data streams between these docks. That this process, called the deployment process, can be automated has been shown [10]. A dock manager has to be created in order to allow multiple docks per host. Because the best place to deploy a dock is dependent on the available and used resources of the available hosts, the ViFramework will monitor this information for all available hosts. To extract this resource information, tooling will be developed to extract CPU, memory and network information from the hosts.

How the framework interface is presented to the system engineer is irrelevant, so a webserver will not be implemented. The system engineer will have access to all information and controls at the host containing the distribution manager.

The ViFramework defines a dock repository which contains all available docks. Although having a distribution manager that is capable of installing any dock on any host at runtime is in general desirable, it is a large step to make from the current situation and is not of much use for ViNotion because their applications use a more or less fixed dock distribution. In the scope of this thesis docks will be defined at host-level and the implementation of a dock repository is left for future work.

Before implementation, a framework design is created based on the framework requirements. The requirements are extracted using four example application scenarios which the framework should support.

### 2.3 Scenarios

In order to extract software requirements for the implementation four scenarios are examined:

- PTZ camera scenario
- Failover scenario
- Expansion scenario
- Internet scenario

These scenarios cover different aspects of what the framework should support. An overview of all requirements extracted from these scenarios is listed in Section 2.5. Besides being used for requirement extraction which is their primary purpose, at least one of these scenario’s will be implemented as a proof of concept. The first scenario presented is the most diverse and by far the most interesting scenario. Therefore, this scenario will be implemented.
2.3. Scenarios

2.3.1 PTZ camera scenario

In this scenario we consider an existing surveillance application developed by ViNotion which includes a Pan Tilt Zoom (PTZ) camera. It consists of multiple components but runs as a complete system on a single host. This application requires video streaming as well as some text-based data connections which makes it an interesting case. The components that can be distinguished in this application are:

- **Static camera**: To be used as object detector.
- **PTZ camera**: To extract object specific information.
- **Video content analyser**: To analyse video feeds with the purpose of detection.
- **User interface**: To present video and generated data.
- **Rule based engine**: To generate control events for the PTZ camera and data for the user interface.

![Diagram of PTZ camera application](image)

Figure 2.5: PTZ camera application

The current setup is depicted in Figure 2.5. The static camera is used for object detection and tracking. When an object is detected by the static camera, the PTZ camera can be used to zoom in on the target and extract more object-specific information. The video stream from the PTZ camera could for example be used for facial recognition on the object zoomed into. Another example is extracting license plate information from detected cars. The PTZ camera is controlled automatically using the data from the analysis component. The user can at any time connect to the interface component and inspect the incoming video streams and application generated metadata. Optionally the user can take manual control of the PTZ camera. It must be possible to install these components on different hosts and setup the connection between them using the distribution manager.
Distribution

In order to determine the requirements for distributed deployment of this application an example deployment scenario is created spanning four hosts. It is important to process the video streams from the cameras close to the source in order to reduce network traffic. Therefore, the video analysis will be done on the hosts that are connected to a camera. That leaves one host for the rule-based engine and one host for user interface. In this scenario the distribution manager is located at the host where the user interface is located but in principle it could be located on any host. This generates the following deployment scenario:

- **Host-1**: Static camera & Video content analyser
- **Host-2**: PTZ camera & Video content analyser
- **Host-3**: Rule-based engine
- **Host-4**: distribution manager & User interface

The described deployment scenario is depicted in Figure 2.6. Connections between framework entities are left out for the sake of clarity.

![Figure 2.6: Distributed PTZ camera application (Deployment)](image)

From this scenario three data categories for communication can be derived: video stream, object data and control data. Each of these can be specified further by adding (application specific) attributes.

- **Video stream**: QoS parameters can be defined, like frame rate, resolution or compression method.
- **Object data**: Support for user-defined object datatypes like object position, type and size.
- **Control data**: Control data format. Here: PTZ camera instructions.
Object- and control data that is to be communicated will vary among applications. Furthermore, fast and easy instrumentation of new components is requires. Therefore, a generic and flexible mechanism for the specification of user defined application specific data formats must be developed.

Requirements

From the PTZ camera scenario and the sketched distribution scenario the following requirements are derived:

- A component can be extended with network functionality in order to create a dock.
- The distribution manager has knowledge of which hosts are available and which are not.
- The distribution manager can deploy docks on available hosts.
- A dock can publish the interfaces it requires.
- A dock can publish the interfaces it provides.
- A dock can have multiple required and provided interfaces.
- The distribution manager can match provided interfaces with requested interfaces.
- Local Area Network (LAN) communication must be supported
- Communication can be set up between two docks with matching interfaces.
- User defined and application specific data formats can be specified.
- Communication QoS properties can be set.

2.3.2 Failover scenario

This scenario describes what happens if a host, currently in use by the framework, would crash. Consider a surveillance application consisting of multiple IP cameras. No host is needed to capture a video stream from such a camera. Instead, the video streams generated by these cameras can be directly accessed using a dedicated IP address. A number of three cameras is chosen, although the exact number of cameras is irrelevant. For each camera there is a corresponding host carrying out analysis, which takes almost all of the host’s resources. We assume that all hosts run the same kind of analysis. When one of the
analysis hosts detects a pre-defined event this is communicated to the server which decides whether and how the user of the system should be notified.

The system is illustrated in Figure 2.7. In this setting a spare host is available which is initially not doing anything. When one of the hosts fails for one reason or another the distribution manager will be notified. The distribution manager is not depicted here but resides on the server. So when it notices the disconnection of (for example) host 3, it will try to perform failover. The docks on the failed host will be re-instantiated on the spare host and the video stream that ran through the failed dock is re-routed through the newly instantiated one. A remote power switch could be used to boot the spare host on demand to save power when the host is not needed. The user of the application will be notified that one of the hosts has failed and has been successfully replaced by the spare host. The user should take care a new spare host is arranged. The new situation is depicted in Figure 2.8.
2.3. Scenarios

Requirements

In addition to the requirements defined earlier, this scenario implies the following requirements:

- Communication between two docks can be removed.
- The distribution manager keeps track of current connections.
- When an error occurs, the distribution manager will be notified.
- The distribution manager can re-route a data stream when the host that forwards that stream fails.

2.3.3 Expansion scenario

Considering a system consisting of surveillance IP cameras of the same type, each covering their own zone, a two-camera system can be depicted like Figure 2.9. Here Host 1 performs an analysis operation on all its incoming streams and passes on its results to a Capture Database on the Server. In addition, Host 1 passes information about its resource usage to the distribution manager, which is places on the server for convenience. Only host-wide resource usage has to be monitored, therefore one framework entity extracting this information per host is sufficient. Because each host has one dock manager that manages all docks on a host, this is the best place to integrate resource monitoring.

When another camera is added to the system the distribution manager can calculate the amount of resources it would take to analyze the stream coming from this camera on Host 1. In this example, analyzing one stream takes about 30% of the host’s resources, so if a system engineer adds another camera this will not overload the system, but adding yet
When four or more cameras are required an extra analysis host will have to be added. The distribution manager will learn about the new host and automatically connect the fourth camera to it. The new situation is depicted in Figure 2.10.

Depending on the policy defined at the distribution manager more changes to the dock allocation could be made. When for example a load balancing policy is in place, one of the video streams currently being analyzed by Host 1 could be re-routed through Host 2 which would result in the situation shown in Figure 2.11.

In this scenario, CPU-load is chosen as the scarce resource, but memory usage could also be the bottleneck and should therefore also be passed on to the distribution manager. Another resource that could be a bottleneck, especially when more cameras and analysis hosts are added is network bandwidth. When, for example, all hosts are located on a 10 Mbit local network, a few high-quality video streams could already overload the network. The distribution manager should keep track of bandwidth usage and report any problems or when possible solve them. It might be possible to reduce the quality of the video streams used in order to reduce bandwidth usage. Therefore, a minimum QoS is required in order to perform correct analysis.
2.3. Scenarios

Requirements

In addition to the requirements defined earlier, this scenario implies the following requirements:

- A host can contain multiple docks.
- All docks on a host are managed by a dock manager.
- The dock manager knows the resource capacity of the host.
- The dock manager monitors resource usage of the host.
- The dock manager can forward resource information to the distribution manager.
- The distribution manager can calculate the network load.
- The distribution manager will keep profile information of docks. (e.g. CPU usage)
- A load distribution policy can be defined and executed at the distribution manager.
- The distribution manager can produce a warning and refuse to deploy when the system engineer tries to deploy a dock to a host which cannot handle that dock.
- The distribution manager can change application parameters in order to change network load.
- A dock can set minimum QoS requirements of the required data stream.
- A dock can publish the QoS parameters it supports.

2.3.4 Internet scenario

The surveillance application shown in Figure 2.12 consists of a server host running a two-step analysis on one video stream coming from a stationary camera. The result of this analysis is a modified video stream that can be watched at a user interface host stationed anywhere on the Internet. The server side of the application typically runs on a Linux operating system, whereas the user interface should run on any platform (at least Linux and Windows). To keep maximum flexibility the server side should also be capable of running on a Windows platform.

A TCP/IP connection could be used to facilitate a communication channel between the two analysis docks. Although a TCP/IP connection to localhost is quite fast, using shared
memory for communication is expected to significantly improve performance. Therefore, local communications should use local shared memory.

The ViFramework must be able to connect hosts outside the local network but connected to the Internet in order to distribute the software over a larger physical space and to allow remote control from anywhere required.

Requirements

Using this scenario the currently defined requirements can be extended with:

- The distribution manager should be able to expand the application crossing the borders of the LAN using the Internet.
- Communication between two docks on the same host should be done using local shared memory. (Managed by a dock manager)
- The developed software should in principle be platform independent.

2.4 Non-functional requirements

In addition to the requirements that follow from the presented scenarios, there are requirements imposed by ViNotion:

- Use of open source libraries: ViNotion must be allowed to adapt the software used for their own needs.
- C++: This is the main programming language used by ViNotion and so it must be used to implement this software.
• **Logging**: In order to enable easy runtime debugging, all relevant debugging information must be propagated to the highest level.

### 2.5 Requirements overview

In this section requirements defined in the previous section are grouped according to the framework entity they belong to, and prioritized. Each requirement is given a unique identifier for easy referencing. Priorities are based on input of the stakeholders and on the approach chosen in Section 1.4. For prioritizing the requirements the MoSCoW method [12] is used:

- **Priority 1**: The software **must** fulfill the requirement.
- **Priority 2**: The software **should** fulfill the requirement if possible.
- **Priority 3**: The software **could** fulfill the requirement when nothing else is affected.
- **Priority 4**: The software **won’t** fulfill the requirement this time but **would** like to fulfill it in the future.

Table 2.1 lists the general requirements of the ViFramework. Dock requirements are listed in Table 2.2. Table 2.3 shows dock manager specific requirements and in Table 2.4 the requirements of the distribution manager are shown. Finally, Table 2.5 gives the requirements of communication used in the framework.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN-01</td>
<td>The software is in principle platform independent.</td>
<td>1</td>
</tr>
<tr>
<td>GEN-02</td>
<td>The software runs on Linux.</td>
<td>1</td>
</tr>
<tr>
<td>GEN-03</td>
<td>The software runs on Windows.</td>
<td>3</td>
</tr>
<tr>
<td>GEN-04</td>
<td>All software is written in C++.</td>
<td>1</td>
</tr>
<tr>
<td>GEN-05</td>
<td>Only open source libraries are used.</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: General requirements
Components are extended with network interfaces, creating docks.

A dock publishes its required interfaces

A dock publishes its provided interfaces

A dock can have multiple input and output interfaces

On error, a dock reports this to the Distribution Manager.

A dock sets minimum QoS requirements

A dock publishes the supported output QoS parameters.

A host can contain multiple docks.

All docks on a host are managed by a Dock Manager.

The Dock Manager knows the resource capacity of the host.

The Dock Manager monitors resource usage.

The Dock Manager forward resource information to the Distribution Manager.

Communication between two docks on the same host uses local shared memory.

Table 2.2: Dock requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOC-01</td>
<td>Components are extended with network interfaces, creating docks.</td>
<td>1</td>
</tr>
<tr>
<td>DOC-02</td>
<td>A dock publishes its required interfaces</td>
<td>1</td>
</tr>
<tr>
<td>DOC-03</td>
<td>A dock publishes its provided interfaces</td>
<td>1</td>
</tr>
<tr>
<td>DOC-04</td>
<td>A dock can have multiple input and output interfaces</td>
<td>2</td>
</tr>
<tr>
<td>DOC-05</td>
<td>On error, a dock reports this to the Distribution Manager.</td>
<td>3</td>
</tr>
<tr>
<td>DOC-06</td>
<td>A dock sets minimum QoS requirements</td>
<td>2</td>
</tr>
<tr>
<td>DOC-07</td>
<td>A dock publishes the supported output QoS parameters.</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.3: Dock manager requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM-01</td>
<td>A host can contain multiple docks.</td>
<td>3</td>
</tr>
<tr>
<td>DOM-02</td>
<td>All docks on a host are managed by a Dock Manager.</td>
<td>3</td>
</tr>
<tr>
<td>DOM-04</td>
<td>The Dock Manager knows the resource capacity of the host.</td>
<td>2</td>
</tr>
<tr>
<td>DOM-03</td>
<td>The Dock Manager monitors resource usage.</td>
<td>2</td>
</tr>
<tr>
<td>DOM-05</td>
<td>The Dock Manager forward resource information to the Distribution Manager.</td>
<td>2</td>
</tr>
<tr>
<td>DOM-06</td>
<td>Communication between two docks on the same host uses local shared memory.</td>
<td>3</td>
</tr>
</tbody>
</table>
### Table 2.4: Distribution manager requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM-01</td>
<td>The Distribution Manager has knowledge of which hosts are available and which are not.</td>
<td>1</td>
</tr>
<tr>
<td>DIM-02</td>
<td>The Distribution Manager can deploy docks on available hosts.</td>
<td>1</td>
</tr>
<tr>
<td>DIM-03</td>
<td>The Distribution Manager matches provided interfaces with requested interfaces.</td>
<td>1</td>
</tr>
<tr>
<td>DIM-04</td>
<td>The Distribution Manager keeps track of current connections.</td>
<td>1</td>
</tr>
<tr>
<td>DIM-05</td>
<td>The Distribution Manager can re-route the data stream when a host forwarding the stream fails.</td>
<td>2</td>
</tr>
<tr>
<td>DIM-06</td>
<td>The Distribution Manager calculates network load.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-07</td>
<td>The Distribution Manager keeps profile information of docks.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-08</td>
<td>A load distribution policy can be defined.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-09</td>
<td>The Distribution Manager produces a warning and refuses to deploy when the system engineer tries to deploy a dock which the application cannot handle.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-10</td>
<td>The Distribution Manager can change application parameters in order to change network-load.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-11</td>
<td>The Distribution Manager can make connections over the Internet.</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table 2.5: Communication requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM-01</td>
<td>A communication can be set up between two docks with matching interfaces.</td>
<td>1</td>
</tr>
<tr>
<td>COM-02</td>
<td>User defined and application specific data formats can be specified.</td>
<td>2</td>
</tr>
<tr>
<td>COM-03</td>
<td>Communication QoS properties can be set.</td>
<td>2</td>
</tr>
<tr>
<td>COM-04</td>
<td>A communication between two docks can be removed.</td>
<td>1</td>
</tr>
<tr>
<td>COM-05</td>
<td>LAN communication should be supported.</td>
<td>1</td>
</tr>
<tr>
<td>COM-06</td>
<td>Internet connections can be setup.</td>
<td>2</td>
</tr>
</tbody>
</table>
Chapter 3

Framework Architecture

In this chapter the high-level ViFramework architecture is presented, but first the definition of components in the scope of the ViFramework will be discussed.

3.1 Component definition

Before the instrumentation of software components can be described, a more exact definition of components must be given. For this an underlying software component model is defined. This model defines what components are, how they can be constructed, composed and destroyed.

As stated by Kung-Lau and Zhen Wang [13] current component models can roughly be divided into two categories, namely models in which components are objects and models where components are architectural units. From the viewpoint of the ViFramework, where the focus lies on creating modular components that can be a part of a more or less fixed pipeline of video processing algorithms, the latter is the obvious choice.

When considering video streaming applications the general structure of the applications’ architecture is more or less fixed. As P. Westerink and F. Schaffa [14] noted, this general architectural structure of multimedia streaming applications comprises an input, one or more processing steps and an output. In their paper they present a flexible framework for building multimedia streaming applications.

The ViNotion software repository does not consist of pre-compiled components “on the shelf”. There are a number of C++ libraries available consisting of video processing algorithms like object detection and tracking. Also a number of auxiliary libraries are available that, for example, handle RTP video streaming. Taking this as a starting point a component
3.1. Component definition

In the ViFramework is defined at source code level, rather than being a binary executable. In the design phase a programmer collects the library calls he needs and creates a method. A component class can also be created. From now on this class or method will be called the “Component”. The created components need to be instrumented to transform them into a docks.

Life cycle processes include all activities of a product or a system during its entire life. Some well known sequential life cycle models are the waterfall model and the spiral model. According to I. Crnkovic et al. [15] the following activities can be identified in any life cycle process model:

- Requirements analysis and specification
- System and software design
- Implementation and unit testing
- System integration
- System verification and validation
- Operation support and maintenance
- Disposal

With the use of CBSE sequential models are no longer feasible. Existing models need to be extended or changed in order to cope with component specific activities. For example the “component development” process and the “system development with components” process should be distinguishable. S.A.Fahmi and H.J.Choi [16] discuss three life cycle models designed for CBSE.

The ViFramework is not involved in all activities that can be found in these component-based life cycle models mentioned above. Currently, the only activities considered are design, implementation and integration. When focusing on the life cycle of a component within the application a similar life cycle appears. As proposed Lau and Wang [13], assuming component development is a sequential process rather than an iterative one and assuming no testing is necessary, an idealized life cycle of a component can be defined to consist of the following phases:

- **Design**: The phase in which components are constructed, cataloged and stored in a repository for future retrieval.

- **Deployment**: The phase in which components are retrieved from the repository and compiled to binary code. The compiled components can be composed into an application.
• **Run-time:** The phase in which the components of a system are instantiated with data and then executed.

In the ViFramework the design phase is where existing components are instrumented in order to become a dock. There are two options when deploying a dock on a host covered by the framework. The first option is to compile the dock once for each platform in order to create a binary that can be uploaded to each host that requires it. The second option is uploading the source code of the dock to each hosts requiring it, and compiling the code locally on each of these hosts. The ViFramework uses the latter, and because the implementation of the dock repository is left for future work this means that in the deployment phase only docks that are pre-compiled at the host are available. Therefore, the deployment phase is no more than composing the compiled docks in order to create an application. When all components are composed in this phase, they can be instantiated and executed which is the run-time phase. The destruction of a component amounts to removing the component instance.

Lau and Wang define four categories of software component models based on composition techniques. When looking at the composition, the underlying model of the ViFramework should be a category 3 model, named “Deployment with Repository”. This means that in the design phase new components can be added to the repository, but they cannot be retrieved from it, implying no composites can be formed in this phase. Only in the deployment phase components can be composed. For the ViFramework, composition at design time is not very useful because when composed at design time the components cannot be decomposed in the deployment phase in order to be distributed over multiple hosts, which is the whole point of the framework. A software component model that uses the same technique and is therefore also in this category is JavaBeans [17].

![Figure 3.1: Idealized life cycle: Category 3](image)

The idealized life cycle of category 3 components is depicted in Fig.3.1. In the ViFramework docks will be what components are for regular component frameworks, a unit of deployment and composition. In the design phase Docks are created and put in a repository, compiling
them on the hosts where they need to be available. After that, in the deployment phase, an assembly tool (distribution manager) can compose the docks by setting up network connections that can be used by the docks’ interfaces. When a network connection is created for all interfaces of a dock it is said to be “deployed”. When deployed, the dock can be instantiated and started which is the run-time phase. All started docks together form the application.

As mentioned before, when we look at component semantics the variant where components are architectural units should be chosen. Here components are modular units of computation connected in an architecture. A component should have well-defined interfaces and be replaceable within its environments. The component syntax of the ViFramework will be a programming language (C++) combined with an Interface Definition Language (IDL) which will be discussed in Chapter 6.

3.2 Framework contents

The ViFramework consists of a static and a dynamic part. The static part of the framework is a library that is to be used to create new instrumented components. This library can for example be used to define interfaces to send processed video frames to. Tools to (partially) automate this process might be developed but at least the procedures to create a new dock will be described. In Fig 3.1 this is the design phase.

The dynamic part of the framework consists of a run-time system that runs on each host. Hosts will run a process that is able to instantiate new docks (dock manager). There are no other processes capable of instantiating docks. One of the hosts will run a different process, possibly next to the dock manager process, that is capable of controlling all other hosts in order to manage connections between instantiated docks (distribution manager). A user interface will also be provided here.

Figure 3.2 depicts the framework architecture. Ideally each dock manager is the only process on a host, manages zero or more dock instances which all have zero or more interfaces. The interfaces are instances of interfaces types included in the static part of the ViFramework.

Dock designers should not be concerned with network programming so the framework should abstract from the network. Besides that the framework should also abstract from the operating system giving room for platform independence. A high level class diagram of the framework is depicted in Figure 3.3. Each host contains a dock manager which has a number of available dock types, which are the dock types that are compiled on this host. If a dock repository was implemented this would be the list of available docks received from this dock repository. Also a list of deployed docks is kept, in the context of the ViFramework
this means that docks are part of the application and are ready to be connected to other docks.

Each dock has a number of interfaces that are used to send or receive data. The exact definition of these interfaces is discussed in Chapter 6. The components that a dock contains in the diagram are in fact all libraries which the dock uses to provide its functionality including “glue code”. This could also be included in the dock, but keeping the functionality of the dock separated from the control is a cleaner solution.
Figure 3.3: High-level class diagram of the ViFramework
Chapter 4

Framework Design

This chapter fills in the high-level framework architecture presented in the previous chapter. It defines the dock, the dock manager and the distribution manager framework entities in more detail.

4.1 Component instrumentation

Each component that is to be used in the distributed application needs to be equipped with network capabilities, transforming these components into docks. Preferably, changes to the components are kept to a minimum. Because components in the ViFramework are a sequence of one or more library calls, room is created for third-party libraries to be instrumented. For example, a third-party library that can perform color filtering on a video frame, could be easily instrumented by calling the corresponding library method on each frame received. It is not feasible to make changes to third party library’s code in order to fit it into the ViFramework. Therefore, the possibility of an intrusive instrumentation (by changing component code) is eliminated. Defining some requirements for these components on the other hand is a possibility.

When considering video streaming applications as is done by Westerink and Schaffa [14], three component roles can be distinguished:

- **Input**: Components that capture video data from an input source (e.g. a camera, a file or an Internet stream), convert it to a common internal format, after which it can be offered to an interface.

- **Processing**: Components that can be used to read video data in the common internal format from an incoming interface, process the video stream before forwarding it to
an outgoing interface in the same format.

- **Output**: Components that accept the common internal format from an interface and convert it to an output format which can, for example, be a display or a video file.

Typically, processing components can reside on any host, whereas input and output components need additional hardware in order to fulfill their task and are therefore located in the proximity of these devices (i.e. a camera or video display).

The usage of a common internal video format requires knowledge of this format within all components. Ideally, components use the same internal file format. The current version of the ViFramework uses ViNotion’s image class. This implies that third parties developing new components need to use this format, or at least know how to convert it to another format.

As opposed to the work of Westerink and Schaffa [14], components are not limited to one incoming and one outgoing interface. Furthermore, different datatypes are possible. Furthermore, the way data is offered to the interfaces might vary. A component may, for example, generate a video stream and on request return a frame counter value (stream-based versus event-based). A choice between pull-based and push-based communication also has to be made. Therefore, each component must define two interface lists; a list of interfaces it provides, and a list of interfaces it requires. The interfaces are defined in more detail in Chapter 6.

Component parameters that are needed to configure the functionality should be known to the framework. An example of such a parameter is a file path needed by an input component that reads from a file. Another example is a component that captures a video stream from a camera. For that purpose, it will need additional parameters enabling camera calibration.

One of the behavioral rules implied by the ViFramework is that the component must be interruptible, which means the component should have some kind of interruptible processing loop. When the component is required to stop, this processing loop should stop and all threads that might be spawned in the components should be interrupted.

Another requirement is that a component should not conflict with other components running on the same host. This can for example happen when a component requires mutual exclusive rights to a system resource. Running a component on a host should not prevent another component from running on the same host.
4.1.1 Dock responsibilities

When instrumented, a component is wrapped in a so-called dock. This dock provides network functionality for the included component. The dock is also responsible for correctly starting and stopping the component when requested, and for setting the required component parameters. Error events received from the component or generated anywhere else in the dock should be forwarded to the dock manager when unresolvable. A graphical representation of the dock structure is shown in Fig 4.1.

![Figure 4.1: Streaming component wrapped in a dock](image)

The dock designer must ensure that the interfaces it uses are well-defined. Also the service, or services the dock provides to the framework, must be known by the distribution manager. These interfaces and the dock functionality will be published to the ViFramework, but this is not a responsibility of the dock, the dock manager is responsible for this. This is possible, because the dock manager will instantiate the dock objects and keep a pointer to it. Therefore, the dock manager has direct access to the docks’ properties. This also means that all framework interfaces to the control part of the dock are direct function calls from or to the dock manager.

An additional dock responsibility is the configuration of QoS properties. Especially for video streaming, QoS is important. Sending uncompressed video data generates a lot of network traffic, which might exceed the network capacity. On the other hand, most VCA applications do not need the highest video quality to function properly. Docks can have multiple output interfaces, which each can have their own QoS properties. Configuring these QoS parameters is done on interface-level. This subject will be discussed in detail in Chapter 6. A dock can set the minimum QoS level needed by its input interfaces. When connected to an output interface QoS negotiation will be performed. This comprises checking the interfaces for compatibility and corresponding QoS properties. When this QoS negotiation succeeds, the QoS properties of the output interface are set to the negotiated values. It is the responsibility of the docks to facilitate the provided service at the negotiated QoS level. When the required QoS cannot be provided for any reason a warning should be
4.1. Component instrumentation

A more detailed description of the QoS negotiation is given in Chapter 6. How the required QoS level should be met depends on the component. The dock could change the component’s parameters if these kind of parameters are available, but a dock could also do processing on its own. Because a dock can provide a video stream to multiple other docks, each with its own requirements, it must be possible to set QoS properties on a connection basis. As an example, consider a dock that reads video from a camera and sends it to two other docks, one on the LAN that performs objects tracking on the video feed, and another that is located outside the LAN for remote monitoring of the stream. In this case the tracking dock should get the video stream at the highest possible QoS level, whereas for the monitoring dock the stream could be downscaled before sending it in order to save bandwidth.

Coupling the dock-interfaces to the component and achieving the required QoS is up to the programmer that will “dock” the component, although some general practice will be proposed.

4.1.2 The Dock class

In order to fulfill the dock responsibilities a dock base class has been designed, which is presented in Figure 4.2. This class contains all basic dock functionality and when implementing a dock for a specific component its implementation must inherit form this base class. Each different derived dock class is a different dock type. The derived docks do not need to have knowledge of their type. When looking at Figure 4.1 this base class takes care of all interfaces from and to the control part of the dock.

Class attributes:

- **id: int**
  Framework wide identifier, provided by the distribution manager.

- **functionality: string**
  Short description of the service or services this dock provides.

- **status: string**
  The current status of this dock. A state diagram of possible dock-states is depicted in Figure 4.3

- **providedInterfaces: [Interface]**
  A list of output interfaces that the dock provides to its environment.

- **requiredInterfaces: [Interface]**
  A list of input interfaces that the dock requires from its environment.
Figure 4.2: Dock Class

- **parameters: [Parameter]**
  A list of parameters that can or need to be set by an entity outside this dock. Typically, these parameters are tuples consisting of a parameter name, a parameter value and optionally a default value.

**Class methods:**

- **Constructor(in dockId: Uint)**
  Creates a dock object with the provided dockId.

- **bindProInterface(int:Interface)**
  Bind the provided interface to an interface of another dock. Depending on the kind of connection that has to be made, additional parameters must be supplied.

- **unbindProInterface(int:Interface)**
  Unbinds the provided interface.

- **bindReqInterface(int:Interface)**
  Bind the required interface to an interface of another dock. Depending on the kind of connection that has to be made, additional parameters must be supplied.

- **unbindReqInterface(int:Interface)**
• setQos(qos: Qos, int:Interface)
  Sets the QoS level of an interface, discussed in more detail in Chapter 5.

• setParameter(index: UInt, value: String)
  Sets the value of the parameter with the given index to the provided value.

• start()
  Starts dock operation (e.g. start streaming video) by executing controlLoop() in a different thread.

• controlLoop()
  This method is executed in a different thread when the dock is started. It is to be implemented by the dock programmer by calling the component here.

• stop()
  Stops dock operation by interrupting the controlLoop() thread.

• reportError()
  When the component contained in this dock raises an error that cannot be handled by the dock, this should be reported to the dock manager which will forward it to the distribution manager if necessary.

Interfaces are classes taking care of communication at a QoS level to be set by the dock. These interfaces are defined in Chapter 5. Also the binding and unbinding of these interfaces will be discussed in more depth in that chapter.

The status of a dock is important because some actions can only be performed if the dock has a certain status. For example a dock which does not have all its interfaces bound will not be allowed to start. There are three possible states a dock can be in; Instantiated, Deployed and Active. The docks state diagram is depicted in Figure 4.3.

When Non-existent, there is no dock object so there also is no dock status attribute. On instantiation the status is set to Instantiated. Then the interfaces of the dock should be bound. Every time an interface is bound it is checked whether all interfaces are bound now. Only if this is the case the dock status is set to Deployed by the dock manager. Only a dock that has status Deployed can be started after which the status will be Active. Unbinding an interface of a started dock is not allowed. This implies that the status will always be Deployed if a dock is stopped. If an interface is unbound and the dock status was Deployed the dock status will become Instantiated because at least one interface is unbound.
4.1.3 Instrumentation

Instrumentation is the process that involves the following actions:

1. Create a new Dock class as a derivative from the Dock abstract base class.
2. Add needed interfaces to the dock.
3. Configure the added interfaces.
4. Construct a list of required parameters.
5. Include the component.

Then it is up to the dock programmer to write code that connects the component to the interfaces of the dock. This “glue code” will reside in the controlLoop() method which is a pure virtual function and therefore needs to be reimplemented for each derived dock. To keep the loop-nature of the controlLoop() method, only returning library calls may be used. When non-terminating library calls are needed, an interruptible thread needs to be spawned for this. It is up to the dock programmer to make sure this thread is correctly interrupted when requested.

Using the interface instances, the component instance and the defined parameters, the programmer must construct a sequence that gives the dock its desired functionality. An example of a sequence is:

1. Create a frame buffer
2. Read frame from a required interface and store it in the frame buffer.

3. Flip the image in the frame buffer.

4. Write the flipped image from the frame buffer to a provided interface.

Only the flipping of the frame is a component code call. The reading and writing of the frame from the interfaces and the creation of the required frame buffer is "glue code" because it is not part of the component or the framework. This sequence can be put inside a loop that runs until an interruption request for this thread has been received. The creation of the buffer does not need to be inside the loop because it has to be instantiated only once. When the control loop is interrupted, all additional threads spawned in the control loop should be interrupted as well, which can be achieved by using exception-capturing interruption requests.

All this gives the controlLoop() method a general structure of:

- **Initialization**: All required variables are declared, application parameters are collected. Furthermore, QoS properties can be collected from the interfaces.

- **Functionality loop**: Here the real work is done, until an interrupt is called or an error occurs.

- **Cleanup**: After the interrupt call (or error), all objects and threads created in this method are destroyed again.

When the new derived dock is completely implemented, it can be added to the dock managers’ available docks list after which it can be instantiated and used to form an application. The dock programmer must make sure that on destruction all instantiated objects are destroyed as well.

### 4.2 Host management

Having no dock repository means that all docks required on a host need to be pre-installed. Not every host needs to have the same collection of pre-installed docks. An entity is needed to manage all docks running on the host. For this purpose the dock manager is created. Docks will be instantiated by the dock manager which will keep a pointer to this dock. Typically, the dock manager will load the available dock types with their corresponding interfaces from a configuration file.

The choice to let the dock manager instantiate dock objects results in a single process per host solution rather than having each dock run in its own process. The active part of each
dock runs in a different thread to prevent interference and to make maximal use of the available CPU cores. An advantage of using this construction is that the dock manager has direct access to all the docks it manages and no inter-process constructs have to be used. If this dock manager process is the only active process on a host, which is preferred, it can make use of all the system resources which makes resource management on the dock manager-level possible.

This is also the reason docks are not responsible for communicating their functionality, interfaces and properties to the framework. The dock manager has direct access to all this information and can therefore forward this information to the distribution manager.

A disadvantage of this single process per host construction is that dock implementations must be very robust. An error in a dock implementation could crash the whole dock manager. So good error handling is required. It is also more difficult, although not impossible, to add new dock types on the fly. Currently, when a new type of dock is added, the dock manager needs to be recompiled because the new dock class has to be included. This is no problem, because of the choice to leave the implementation of the dock repository for future work. In the current version of the ViFramework docks can only be added at compile time.

4.2.1 Dock manager responsibilities

Primarily, the dock manager is responsible for reporting the host’s presence to the distribution manager. On initialization, the dock manager will create a list of available dock types. When the distribution manager learns about the host’s presence, it requests host information. The dock manager should pass the list of available docks types to the distribution manager. If a dock manager goes offline for whatever reason, the distribution manager is informed about this. The network substrate is responsible for this and is discussed in detail in Chapter 5.

After initialization the dock manager is passively waiting for remote procedure calls from the the distribution manager, which implies the dock manager is responsible for receiving and processing these remote procedure calls. These remote procedure calls range from instantiating new docks to reporting relevant information to the distribution manager. After instantiation, a dock can be started, stopped and finally destroyed again.

Some errors raised by a dock can be handled by the dock manager itself, others need to be handled by the distribution manager. Therefore, the dock manager should be able to forward encountered errors to the distribution manager.

Besides providing an interface between the distribution manager and the docks on its host, the dock manager is responsible for monitoring host resources. On demand it should provide
relevant resource information to the distribution manager. The most interesting metrics are CPU utilization, memory usage, and network usage. When there is a resource shortage this should be reported to the distribution manager as an error or warning.

4.2.2 The DockManager class

The DockManager class is developed to take care of the responsibilities defined above. The class diagram is depicted in Figure 4.4.

<table>
<thead>
<tr>
<th>DockManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>+availableDockTypes: [DockType]</td>
</tr>
<tr>
<td>+instantiatedDocks: [Dock*]</td>
</tr>
<tr>
<td>+Constructor()</td>
</tr>
<tr>
<td>+inflateDock(dockId:UInt, type:DockType)</td>
</tr>
<tr>
<td>+destroyDock(dockId:UInt)</td>
</tr>
<tr>
<td>+bindProInterface(dockId:UInt, int:Interface)</td>
</tr>
<tr>
<td>+unbindProInterface(dockId:UInt, int:Interface)</td>
</tr>
<tr>
<td>+bindReqInterface(dockId:UInt, int:Interface)</td>
</tr>
<tr>
<td>+unbindReqInterface(dockId:UInt, int:Interface)</td>
</tr>
<tr>
<td>+setQos(dockId:UInt, qos:Qos, int:Interface)</td>
</tr>
<tr>
<td>+setParameter(dockId:UInt, index:UInt, value: String)</td>
</tr>
<tr>
<td>+startDock(dockId:UInt)</td>
</tr>
<tr>
<td>+stopDock(dockId:UInt)</td>
</tr>
<tr>
<td>reportToDManager()</td>
</tr>
</tbody>
</table>

Figure 4.4: DockManager class

Class attributes:

- availableDockTypes: [DockType]
  List of dock types available on this host.

- instantiatedDocks: [Dock*]
  List of pointers to instantiated docks on this host.

Class methods:

- Constructor()
  Creates a DockManager object, fills the availableDockTypes list using a configuration file. The DockType type describes the dock by means of functionality, interfaces and parameters.

- instantiateDock(dockId:UInt, type:DockType)
Instantiates a new dock instance of the given type on this host, which gets dockId as identifier.

- **destroyDock(dockId: UInt)**
  Destroys the dock with framework identifier: dockId.

- **bindProInterface(dockId: UInt, int: Interface)**
  Bind a provided interface of the given dock to a required interface of another dock. Depending on the kind of connection that has to be made, additional parameters must be supplied.

- **unbindProInterface(dockId: UInt, int: Interface)**
  Unbind the given provided interface of a dock.

- **bindReqInterface(dockId: UInt, int: Interface)**
  Bind a required interface of the given dock to a provided interface of another dock. Depending on the kind of connection that has to be made, additional parameters must be supplied.

- **unbindReqInterface(dockId: UInt, int: Interface)**
  Unbind the given required interface of a dock.

- **setQos(dockId: UInt, qos: Qos, int: Interface)**
  Set the QoS level of the interface provided of the given dock. Discussed in detail in Chapter 6.

- **setParameter(dockId: UInt, index: UInt, value: String)**
  Sets the value of the parameter with the given index of a dock to the provided value.

- **startDock(dockId: UInt)**
  Starts the dock with the provided dock ID. Creating a thread in which the controlLoop() method of the dock is run.

- **stopDock(dockId: UInt)**
  Stops the dock with the given dock ID.

- **reportToDManager()**
  Report information to the distribution manager, including host information, available docks, resource usage, and optionally information about docks deployed.

Most class methods are just a matter delegating a command to the dock with the given dock ID. On creation of the dock manager class, the presence of this new dock manager should be reported to the distribution manager. Because it is not yet defined how the distribution manager communicates with the dock managers, the method reportToDManager is, in this stage of the design, a generalization of all methods that are used to communicate with the
4.2.3 Resource monitoring

To enable automatic application deployment, load distribution and host failure recovery, the distribution manager needs information about the available and used resources (e.g. CPU, memory, network bandwidth) on each connected host. Resource information is gathered by the dock managers and forwarded to the distribution manager. This enables resource management on two levels; at host-level and at system-level.

Resource requirements for video content analysis algorithms are often data-dependent [18]. This requires the framework to respond to a sudden increase in resource requirements. If a host cannot meet the resource requirements of its docks, the distribution manager will try to redistribute the application in a more appropriate way.

Although resource monitoring is not the main focus of this project, some superficial statistics will be calculated and send to the distribution manager as a proof of concept. For this project only the CPU load, the memory usage and used network bandwidth of the host will be extracted from the operating system.

In order to do this resource extraction a separate thread will be spawned that periodically extracts resource usage information from the operating system. Knowing the period, the average resource usage over the interval can be calculated.

When a larger number of hosts are used in an application, periodically sending resource usage information to the distribution manager becomes problematic because of network bandwidth usage. Furthermore, the distribution manager might not need resource information from each dock manager at a high rate. Consider CPU load for example, which might peak when a dock is started but will decrease to acceptable levels fast. Furthermore, not all changes are equally interesting for the distribution manager, for example a CPU usage increase from 25% to 26% is in most cases irrelevant.

If every change in resource usage is reported at a high rate for a large number of docks, this will generate a lot of network traffic, and because of that make the ViFramework not well scalable. Larisa Rizvanovic and Gerhard Fohler have proposed a method for coping with this problem in a paper about a framework for real-time resource management for video streaming in networks of heterogeneous devices called The MATRIX [1]. Their idea is to map the resource usage to three static discrete values; low, medium or high as is depicted in Figure 4.5.

Although the idea is potent, a lot of information is lost, and data that is a bit more specific could be needed in order to make good choices in the distribution manager. The
method used in the ViFramework is a bit more extensive than this. The resource extraction method is configurable to fit any application. The usage of all resources is calculated as percentages. First the measurement interval can be chosen. Then there are two thresholds to set. The difference threshold, that determines the minimal change in resource usage that has to be observed before reporting is the first one. When this thresholds is for example set to five percent and the last reported CPU load was 50%, only a value over 55% or under 45% will be reported. The second thresholds is the so-called delay threshold, that determines the minimum duration (in measurement intervals) required before the new value will be reported. Using this low-pass filtering construct, peaks in resource usage can be ignored, where the second thresholds sets the minimal width of the peak that is needed for reporting. This way, also oscillation of the resource graph is avoided.

An example of the resource monitoring using the resource usage abstraction method mentioned above is depicted in Figure 4.6. The left graph depicts a possible CPU load profile. If the difference threshold is chosen to be 5 and the delay threshold is chosen to be 3, the resulting resource graph is depicted in the right hand graph of Figure 4.6.

At the distribution manager the graph of the CPU usage of this host will look like the right hand side of Figure 4.6. This saves a lot of network traffic by only reporting relevant
information about each host. But when needed, still a very exact and high-rate resource usage image can be created by lowering the thresholds and/or the measurement interval.

4.2.4 Future functionality

Besides working with pre-installed docks, it would be desirable that the framework should add and remove docks on-the-fly. This would allow new dock types to be uploaded and used right away. In the current situation, new dock types have to be manually compiled at all the hosts in the covered by the framework. Docks could be send to the dock manager which would compile them, then a construction to dynamically link this newly created dock type to the dock manager is needed. This is left for future work.

Active resource management is also left for future work. The dock manager could be extended with functionality to manage the resource usage per dock. Memory reservation or CPU allocation for each dock could for example be managed by the dock manager providing it with a resource scheduler. Host-wide resource scheduling is left to the operating system, assuming there are enough resources available. When host-wide resources are not sufficient, this will be reported, so on a higher level (i.e. distribution manager level) choices can be made to resolve this problem.

4.3 Distribution management

For deploying and managing a distributed application a distribution manager is created. A user interface for this distribution manager enables the system engineer to setup the required application on multiple hosts without the need to login to each host separately and setup each host on its own. As long as each host has a dock manager running, using the distribution manager interface will give the system engineer adequate capabilities to setup the whole application.

4.3.1 Distribution manager responsibilities

On initialization the distribution manager must announce its existence, triggering available dock managers to send their information to it. With the information received from available dock managers, the distribution manager creates a list of what dock types are available on which hosts. From now on this list will be called the dock table.

Using the user interface of the distribution manager a system engineer can deploy docks on the available hosts. A remote procedure call to the dock manager will be made which
will instantiate a dock of the requested dock type. Whereas for communication between docks it would suffice to let the interfaces define the dock type, for “moving” docks from one host to another, instantiating a dock with similar interfaces on an other host might raise problems. Docks with similar interfaces do not need to provide similar functionality. Therefore, we also need to uniquely define functionality of a dock. This is why each created dock type is assigned a unique identifier and a corresponding functionality string.

The distribution manager has several functions at its disposal to instantiate docks, connect docks to each other if their interfaces match, start and stop docks, disconnect docks and finally destroy them again.

To give the system engineer an overview of the current situation, the distribution manager will provide a method to show the current distribution, showing where, which docks are deployed and what the current status of that docks is.

### 4.3.2 The DistributionManager class

![Figure 4.7: DistributionManager class](image)

#### Class attributes:

- **dockTable: (HostId, [DockType])**
  
  A list of available hosts, with an immutable list of supported dock types per host.

- **instantiatedDockTable: (HostId, [DockInfo])**
  
  A list of instantiated docks, grouped by host, containing status information about
4.3. Distribution management

the instantiated docks including dock status (Figure 4.3) and binding information of its interfaces.

- **connections**: [(UInt, UInt), (UInt, UInt)]
  A list of connections consisting of pairs of (dock identifier, interface identifier) tuples. All information in this table is also recoverable from the deployedDockTable. This table is created for efficiency reasons only.

DockType definition will be discussed further on.

**Class methods:**

- **Constructor()**
  Creates the distribution manager instance, publishes its presence and listens for dock managers to report. Because of the network substrate discussed in Chapter 5, presence has to be published only once.

- **instantiateDock(host: String, type: DockType)**
  Sends a command to the given dock manager to deploy a dock of the provided type.

- **destroyDock(dockId: UInt)**
  Destroys the dock with the given dockId from the application.

- **startDock(dockId: UInt)**
  Starts a dock when if all interfaces are connected.

- **stopDock(dockId: UInt)**
  Stops a dock.

- **addConnection(dock1: UInt, int1: UInt, dock2: UInt, int2: UInt,)**
  Add a connection between two compatible interfaces.

- **removeConnection(dock1: UInt, int1: UInt, dock2: UInt, int2: UInt,)**
  Remove an existing connection between two interfaces.

- **matchInterfaces(dockId: UInt, int: UInt,): [(UInt, UInt)]**
  Return a list of tuples containing dock IDs and interface numbers that are compatible with the provided one.

- **bindProInterface(dock1: UInt, int1: UInt, dock2: UInt, int2: UInt,)**
  Private method used by addConnection() to bind a provided interface to a required one. Depending on which type of connection needs to be setup, additional arguments must be provided.

- **bindReqInterface(dock1: UInt, int1: UInt, dock2: UInt, int2: UInt,)**
Private method used by `addConnection()` to bind a required interface to a provided one. Depending on which type of connection needs to be setup, additional arguments must be provided.

- **handleError(dockId: Uint, errorMessage: String)**
  Handles an error originating from a dock.

- **showDistribution()**
  Shows the current distribution.

The method `showDistribution` is a generalization of all methods that help presenting the current system status to the end-user.

Next to the interface provided to the system engineer the distribution manager also needs to listen to messages from the dock managers. Error messages could arrive, but also acknowledgements of send commands. Because of the network substrate used and discussed in Chapter 5, the distribution manager must implement a number of event handlers that are called by the substrate when such a message arrives. The handling of acknowledgement- or error messages as a response on a command are handled by the corresponding command method (e.g. `startDock()` and `addConnection()`). Other error messages that arrive (e.g. when a dock crashes) are handled by the `handleError()` method.

### 4.3.3 Profiling

The distribution manager will be collecting resource information from the dock managers. This resource information will be collected by the dock managers as is described in Section 4.2.3. With this information the distribution manager can create a profile of the resource usage of a certain dock on a certain host. These profiles can be used by the distribution manager to make good choices on where to deploy a new dock. For instance, the distribution manager could decide to migrate docks from one host to another in order to perform load balancing. In order to do sensible profiling more extensive resource extraction at the dock manager level is required. Because of that, profiling is left for future work.

Combining all entities of the ViFramework results in the complete system as is depicted in Figure 3.3. Using the distribution manager interface, a system engineer can control the whole system.
4.4 Behavioral description

The system engineer can decide what dock types to instantiate on which hosts. Figure 4.8 depicts this process where the distribution manager provides a new dockId and sends the dock manager of the chosen host a command to instantiate a dock of the given type. The distribution manager will wait for an acknowledgement before updating the global system state.

![Figure 4.8: Sequence diagram: Instantiate dock of a given type](image)

Destroying docks works in the same fashion, with the only difference that a dock ID is enough to identify the dock that has to be destroyed. If a dock cannot be destroyed, because it is for example non-existent, the distribution manager will just ignore the command, providing the end-user with a warning, the command could not be executed.

![Figure 4.9: Sequence diagram: Destroy dock](image)

Interfaces of deployed docks can be connected to matching interfaces (matchInterfaces()) of other docks. For this purpose the System Engineer will ask the system to match a provided interface to all other interfaces known, after which he can choose a match to connect.
Connecting means the distribution manager will bind both interfaces of the connection to each other. The connection process is depicted in Figure 4.10. Although the connection process is initiated by the system engineer, this actor is not shown in the diagram for the sake of clarity.

In this case we assume that a port number has to be negotiated. This could be the case when we want to setup a dedicated TCP-channel between two docks in order to communicate a video stream. The host that receives the video stream must decide which port number it will listen to. This port number has to be forwarded to the sending dock. Whether providing addition information in order to setup communication is necessary, depends on the network substrate which will be discussed Chapter 5.

If a connection needs to be removed first the required interface needs to be unbound. When the required interface stopped listening the provided interface can also be unbound. This process is depicted in Figure 4.11.

If all interfaces of a dock are connected, the dock can be started. This could be done automatically by the distribution manager or we could wait for a command from the system engineer. The second case is shown in Figure 4.12. After the command from the system engineer, the system checks whether all interfaces are bound. If this is not the case, an error message will be returned to the system engineer. Starting the dock is done by an
asynchronous start() call to the dock. This call will be acknowledged immediately and in a separate thread the controlLoop() method of the dock will be started.

When the command is given to stop the dock, the thread running the controlLoop() method is interrupted. This is depicted in Figure 4.13. All dock settings are saved so the dock can be restarted directly without having to bind the interfaces again.

If any of the previously defined command fails for one reason or another, an “Error” is returned in stead of an “Ack”. If such an error occurs the system state will not be changed and the end user will receive an error message explaining what has gone wrong. This is not depicted in the sequence charts.
Chapter 5

Network Abstraction

When distributing an application over a (distributed) platform using the ViFramework, the system engineer should not be concerned with handling network connections. The framework should provide location transparency so the system engineer can connect docks independently of their physical place in the network. For this purpose, a good network substrate must be chosen as basis of the frameworks’ middleware. This chapter first defines the requirements of such a network substrate. Secondly, the substrate proposed by van Langen and Opdam [10] is discussed. A number of alternatives are investigated and finally, a choice is made for a substrate that best fits the ViFramework.

5.1 Requirements

In Section 2.5 the requirements of the ViFramework are defined. A number of them relate to the network substrate. In addition to these, some requirements are defined that focus on future extensions of the ViFramework.

To enable compatibility with the software made by ViNotion the software used for network communication should be written in C++ as is all ViNotion software. Platform independence also is an import issue. Nowadays ViNotion uses Linux and Windows components and they are interested in using components for mobile devices in the near future.

Synchronous Inter-Process Communication (IPC) is not crucial for the ViFramework. It could be used for communication between the distribution manager and the dock managers. Because it is important that the distribution manager knows for sure the system state has changes before allowing the next command to be executed, this would be an obvious choice. More important is the support for asynchronous IPC between docks. In general, docks encapsulate computationally intensive algorithms, these should not be blocked by
5.2 OpenDDS

Van Langen and Opdam [10] proposed the use of the openDDS framework as a basis for the ViFramework. OpenDDS has strong points that make it more suitable for use over the other investigated frameworks, although only three candidates were thoroughly investigated in their work. Most of the available frameworks were discarded because of their platform dependencies or the programming language used.

Advantages of the OpenDDS framework are:

1. **Complex dataflows:** OpenDDS provides neutral facilities for networks that contains nodes with fast and slow connections. Extensive QoS parameters allow integration of systems that require different update rates, reliability and bandwidth control.

2. **End-to-end connections:** The ability to set up direct end-to-end connections enables good network performance.

3. **Performance:** As stated by the Object Management Group [19], openDDS has proven to have low latencies and a high data-rates. Furthermore, the framework has proved to be very well scalable.

4. **Dynamic reconfiguration:** OpenDDS is very suitable for dynamic reconfiguration
because the framework is in essence data-centric using the publish-subscribe model which provides location transparency. Using dedicated control topics, auto-discovery of nodes can be implemented as is proposed by van Langen and Opdam [10].

So indeed, openDDS seems like a good starting point for implementing the ViFramework. Unfortunately, there are also some major drawbacks associated with openDDS, especially when the main focus of the framework is filter-like video streaming components, which the most applications of ViNotion are.

Disadvantages of the OpenDDS framework are:

1. **Publish/Subscribe**: The publish-subscribe model used by OpenDDS is not very useful if the topology of the network and the configuration of the application is more or less fixed, which is almost always the case. Point-to-point connections are most of the time sufficient to satisfy communication needs. Furthermore, the openDDS framework is not very suitable for a request-reply kind of communication and for file transfer [19].

2. **Video streaming**: Van Langen and Opdam [10] concluded that using openDDS for video streaming is not feasible.

3. **Footprint**: The OpenDDS library leaves quite a large footprint on the hosts it is installed on and provides a lot of functionality that probably will never be used in the ViFramework.

4. **Flexibility**: OpenDDS is essentially data-centric and forces you to first design the interfaces of your components in an IDL generating component skeletons from that definition. Although this code generation saves some work, the slightest change to the interface will force a recompilation and the skeleton of the components has to be regenerated with the changed interface. Furthermore, openDDS does not provide constructs to communicate dynamic datatypes.

5. **Security**: OpenDDS does not have any communication security features.

6. **NAT-router and firewall traversal**: No facilities to traverse through NAT-routers or firewalls are provided. This makes it difficult to extend applications beyond the borders of the LAN.

From these drawbacks rose the decision to drop the openDDS framework and search for an alternative.
5.3 Alternatives

The advanced component system SOFA 2.0 [20] was created in order to overcome limitations of formerly existing component-based systems. Due to the lack of video streaming support and the service-oriented nature of SOFA 2.0 this component system is considered unsuitable for real-time video streaming applications. Also, this framework is programmed in JAVA, which conflicts with the requirements of this project (GEN-04).

GStreamer [21] is a framework focused on audio and video streaming applications. The framework aims at creating single machine multimedia applications by composing existing components called plug-ins. The framework has no facilities to communicate presence information (DIM-01) of framework entities, which is a main feature of the ViFramework. Furthermore, GStreamer has no built-in support for dynamic reconfiguration (DIM-05).

Westerink and Schaffa [14] propose a flexible framework for building multi-media streaming applications. This framework identifies the general architectural structure of streaming applications and uses this knowledge to create an easy-to-use framework which is used for a number of existing applications. The framework proposed by Westerink is targeted at creating single machine applications from existing components, and therefore not suitable to be used as a networked component framework.

A network substrate for desktop grid computing named Orbweb [2] is presented by Schultz et. al. It describes the effort that is made to extend the XMPP protocol in order to meet the substrate needs. This substrate uses XMPP for NAT and firewall traversal and takes advantage of the available security protocols embedded in XMPP. Orbweb is considered unsuitable because no facilities for real-time applications is available.

Combined with the work done by van Langen and Opdam [10], it can be concluded that there exists no component framework that satisfies all the needs of the ViFramework.

5.4 XMPP

Because existing networked component frameworks do not satisfy the ViFramework’s needs and require extension of an often already large framework, the decision was made to create the ViFramework directly on an application layer communication protocol called the eXtensible Messaging and Presence Protocol (XMPP) [3].
5.4.1 Introduction

XMPP is an open standard for real-time communication formally known as Jabber. It was originally created for messaging applications but because of its modularity and easy extendability it is now being used for all kind of applications. For us, the most interesting of these are voice and video calls as well as lightweight middleware which is exactly what is need for the ViFramework. XMPP communication is based on the eXtensible Markup Language (XML) which makes it simple and very flexible and therefore very suitable for an implementation of the ViFramework.

5.4.2 Advantages

The XMPP protocol is well documented. Moreover, there are clients, servers and libraries available for virtually all platforms and programming languages. Whereas the basic protocol only provides simple messenger functionality such as sending messages, contact lists and presence information, there are many extensions available to the protocol. A lot of these are also adopted by most libraries or, if not, are easy to implement. These extensions range from data-forms to service-discovery and feature negotiation to stream initiation which all are very helpful when implementing the ViFramework. Because of the modularity we only have to add the extensions we need to our application which keeps it lightweight and simple.

The most extensive C++ library available is gloox which is stable, fully documented and platform-independent (supports Windows, Linux, Mac OS X and a couple of mobile operation systems). It supports almost all basic functionality, implements a large number of important protocol extensions and is easy to extend.

Shultz et al. created an XMPP-based network substrate for parallel applications that requires peer-to-peer interaction. They used it for their “Cohesion Orbweb” application which creates a substrate for peer-to-peer desktop grid computing applications. They used XMPP to tackle domain-specific challenges like high scalability, Network Address Translation (NAT) and firewall traversal. A representation of their network structure is depicted in Figure 5.1. NAT and firewall traversal are very important features when communication across the borders of a Local Area Network is needed. XMPP tackles this challenge by tunneling data through the HTTP and SOCKS 5 protocols.

All kinds of communication types can be easily implemented using standard constructs. The protocol distinguishes three types of communication constructs. A “Message” can be used to implement the sending data in a push-based fashion, whereas an “IQ” can be used to setup pull based communication implementing a Get-Result or Set-Result communication. The “Presence” construct can be used for communicating status information of network entities (Clients), but can also be used to implement publish-subscribe like
In contrast to OpenDDS, XMPP provides several levels of security. Because each network entity has to authenticate itself at the host server, spoofing is impeded. Additional security measures used are server dailback and whitelisting which define what servers are granted access. In the connection establishment phase as well as in the communication phase both Transport Layer Security (TLS) and Simple Authentication and Security Layer (SASL) are used which make client-server communication in XMPP inherently secure. As XML data can potentially be routed over an intermediary server that may belong to third parties, XMPP provides end-to-end signaling and object encryption to prevent eavesdropping.

5.4.3 Disadvantages

Unfortunately, there are also some major drawbacks in using XMPP for our application. Foremost is the lack of a peer-to-peer interaction model. Even messages that could be send directly between two clients are relayed by the XMPP server. Adding the fact that XML is very verbose, the server will most definitely become a bottleneck. Especially when sending video streams using an inbound byte stream (making use of the protocol) will put a great load on the server and will double the network traffic.

Fortunately, some extensions to the XMPP protocol exist that take care of these drawbacks. For example, XEP-0138 defines stream compression based on the ZLIB library. Gloox automatically uses this compression when the server supports it which saves network traffic. More importantly, the Jingle extension (XEP-0166) supports signaling for initiating and
managing peer-to-peer media sessions between two XMPP entities. A related protocol extension (XEP-0167) negotiates Real-time Transport Protocol (RTP) sessions between two clients which could be used for video streaming. Unfortunately gloox does not support the Jingle extension yet, although this is planned for the next release (version 1.1). For now, the signaling of media sessions will have to be part of the ViFramework until this is supported by gloox. The RTP connections themselves will have to be set up by the framework anyway because only facilitates the signaling part. From the viewpoint this is an outbound byte stream.

By extending with the ability to setup peer-to-peer connection an efficient, scalable and secure network substrate can be created as is done by Shultz et al. [2].

5.4.4 Summary

A comparison between OpenDDS and XMPP is shown in Table 5.1. The “XMPP Remarks” column in this table either explains how implements an aspect or which protocol extension could be used to implement it.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>OpenDDS</th>
<th>XMPP</th>
<th>XMPP Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Video streaming</td>
<td>-</td>
<td>-</td>
<td>Jingle (XEP-0167)</td>
</tr>
<tr>
<td>Complex data-flow support</td>
<td>+</td>
<td>+/-</td>
<td>MUC / Pub-Sub (XEP-0060)</td>
</tr>
<tr>
<td>P2P communication</td>
<td>+</td>
<td>-</td>
<td>(XEP-0246)</td>
</tr>
<tr>
<td>Verbosity</td>
<td>+</td>
<td>-</td>
<td>XML compression (XEP-0138)</td>
</tr>
<tr>
<td>Flexibility</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>NAT/Firewall traversal</td>
<td>-</td>
<td>+</td>
<td>HTTP / SOCKS5</td>
</tr>
<tr>
<td>Security</td>
<td>-</td>
<td>+</td>
<td>TLS / SASL</td>
</tr>
<tr>
<td>Presence Handling</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Network substrate comparison

Observing that the major problems of the protocol for our application can be solved by using existing extensions or by writing one of our own, we conclude that it is good basis to build the ViFramework on, although now there are no facilities for component management which will have to be implemented. We must for example specify how components should behave within the framework and how their interfaces can be defined.
A flexible framework for distributed components needs to support a wide range of interface types. Preferably, software developers creating new dock types do not need to make changes to the ViFramework itself. This calls for a framework that supports virtually all interface types.

In the ViFramework interfaces have a static and a dynamic aspect. The static aspect of an interface is its signature, which defines what data category is send or received by the interface. This definition is created at design time and stored in an XML document.

The dynamic aspect of an interface provides end-users of the ViFramework with facilities to configure them at deployment time. At deployment time and runtime, interfaces are instantiated objects that are responsibility for sending and receiving data. Communication is done conform the interface definition and configuration and is independent of the network path that has to be traversed.

This chapter will elaborate on interface definition, matching and binding. Furthermore, the used communication principles are discussed.

### 6.1 Interface definition

In the ViFramework interfaces are distinguished by the category of data they facilitate communication for, and whether they are provided or required interfaces. In the scenarios specified in Section 2.3, three data categories are distinguished.

1. Video data
2. Object data

3. Control data

Object data and control data can be communicated using structured text-based datatypes and can therefore be represented using XML. A video stream is a frame-based byte stream and therefore not representable in XML. Another example of data that must be communicated by a byte stream is a file transfer. Since the main focus of the ViFramework is video streaming applications, the emphasis lies on communicating video streams and generated metadata (object data). From this point on, all XML-based data will be referred to as object data. Because file transfers are not considered in this work, the development of four interface types is sufficient, being video streaming interfaces and object data interfaces each with their provided and required variant.

Because flexibility and expressiveness are required when defining interfaces, XML is chosen as an interface definition language. XML is simple and widely used for the representation of arbitrary data structures, which makes it very suitable for the ViFramework. Because XML is chosen to represent all data except video, it is very convenient to also use XML for the interface description itself. It is up to the dock programmer to make sure the interface is implemented conform the interface definition.

An element or object in XML is represented by a number of so-called tags. A tag is a markup construct that begins with "<" and ends with ">". Three types of tags exist:

1. Start tag: `<tagName>

2. End tag: `</tagName>`

3. Empty element tag: `<tagName/>`

An XML element either begins with a start tag and ends with a matching end tag, or it only consists of an empty element. Each element can have a number of attributes defined, these are specified within the start tag or empty element tag.

```
<object>
  <child1 attribute1='x'>
    <grand-child1> abc </grand-child1>
    <grand-child2> def </grand-child2>
  </child1>
  <child2 attribute1='y' attribute2='z'/>
</object>
```

Listing 6.1: XML example

Listing 6.1 lists an example of an object represented in XML. Elements can contain other elements named children, in this listing the root element has two child elements with tag
name `<child1>` and `<child2>` respectively. These children can again contain children. Furthermore, an element can enclose textual data like the "grand-child" elements do in the example XML listing.

In the ViFramework each interface will be defined by one interface element containing only one child that defines the datatype that is to be communicated, which is either a `<videoStream>` or an `<objectData>` element. The contents of the child element depend on the datatype and the preferences of the dock developer. The XML syntax used by the ViFramework is defined by the XML Schema definition listed in Appendix B.

### 6.1.1 Video data

The children of the `<videoStream>` element can define a number of QoS properties with their minimal and maximal values. An example of a definition of a video streaming interface is listed in Listing 6.2. These minimal and maximal values are to be set by the dock implementer and define the QoS range the dock is able to operate in. For example, some VCA algorithms need a minimum resolution to produce meaningful results. These QoS properties are taken into account when matching interfaces. Interfaces that do not have matching QoS properties are not allowed to be connected with each other. Aside from QoS properties, a "topic" attribute can be added to the `<videoStream>` tag in order to identify specific video streams. Only video stream interfaces with corresponding topic attributes will be matched. The matching of interfaces is discussed in more detail in Section 6.2. Listing 6.2 also shows a "current" element in each QoS property, this element will be set at runtime and describes the value this QoS property is set to.

### 6.1.2 Object data

An example of an object data-interface is shown in Listing 6.3. The first and only child of the interface is again the describing element, defining object data will be communicated there. The structure of the object data communicated using this interface is defined within the `<objectData>` element. In this example a surveillance application communicates the number of people counted from one dock to another. The first and only child element of the `<objectData>` element defines the name of the datatype, in this case `<peoplecountingreports>`. What is inside this element is the data representation of the objects to be communicated.

The whole `<objectData>` element will be communicated from dock to dock. The only difference between the interface definition and the data communicated is that the "data-tags" (in this example: `id`, `timestamp` and `count`) are filled with data. The optional "type" attributes of the data-tags are for the dock creators that need to know to what
datatypeto type cast the data to.

The “list” attribute of the <peoplecountingreports> tag specifies that each tag can contain multiple <report> tags when communicated. The “method” attribute of the <objectdata> tag defines whether push- or pull-based communication is required. If acknowledgement of arrived data is required, this can be added to the <objectdata> tag. These attributes are handled by the framework.

6.1.3 XML description

With the use of XML to define interfaces, application designers can easily add new text-based datatypes for communication. Only a new unique name for the datatype has to be found such that the framework can use this for matching. The content of this newly named tag will just be forwarded by the framework and filling and reading the tag before and after communication is up to the dock implementation. Also video streaming interfaces can be easily defined and configured.

In addition to the XML descriptions of the interfaces, also the docks’ parameters and properties are described using XML. Furthermore, dock manager properties such as its identifier and the location of its XMPP server are described this way. This creates a solution where a dock manager can load all information it requires from one configuration.
6.2 Interface matching

At deployment time, a component framework requires methods for component matching in order to automatically setup connections or to aid an end-user in manually setting up these connections by making connection proposals. The distribution manager uses the interface definitions to do matching on a per-interface basis. Each interface instance has an abstraction of its XML definition stored. This definition can be further configured by the end-user.

Using the (configured) descriptions, an interface matching algorithm is created that, given an arbitrary interface, returns all compatible interfaces known to the system. Compatible interfaces are interfaces that can meaningfully be connected to another interface. Provided interfaces can only be connected to required interfaces and the other way around. Assuming the interface that is to be matched is a provided interface, the matching algorithm will first create a list of all available required interfaces. Interfaces are considered available when the dock they belong to has an instantiated status. Furthermore, because it is assumed that a required interface can only be bound to one provided interface, the interface must not be bound already. In contrast to required interfaces, it is possible to connect a provided interface to more than one required interface, as will be explained in detail in Section 6.3. A sketch of the implemented matching algorithm is listed in Appendix E.
6.2.1 Video streaming interfaces

It is evident that a video streaming interface cannot be connected to an object data interface, so they will not be matched. Furthermore, the interface will only be matched to required interfaces that have the same topic attribute or no topic attribute at all. If the video streaming interfaces are compatible up till now, the QoS parameters are matched. For each QoS parameter of the required interface the required QoS level is defined by either a minimum and a maximal value or a set of options. The provided interface must be able to provide a video stream with the required QoS level.

```
<providedInterfaces>
  <interface>
    <videoStream topic='ptz'>
      <fps>
        <min>5</min>
        <max>15</max>
      </fps>
    </videoStream>
  </interface>
</providedInterfaces>
```

```
<requiredInterfaces>
  <interface>
    <videoStream topic='NONE'>
      <fps>
        <min>10</min>
        <max>25</max>
      </fps>
    </videoStream>
  </interface>
</requiredInterfaces>
```

Figure 6.1: Interface matching example: matching video stream interfaces

Figure 6.1 depicts the definition of two compatible interfaces. The required interface has no topic attribute defined, resulting in accepting video streams from provided interfaces with any topic. Also the QoS properties of the interfaces match. The only QoS property considered in this example is frame rate. A frame rate between 10 and 25 frames per second is required, whereas a frame rate between 5 and 15 frames per second can be provided. If a connection between these interfaces is set up, the operating frame rate will be set between 10 and 15, depending on the application preferences or a QoS policy defined by the distribution manager.

An example of interfaces that do not match is depicted in Figure 6.2. In this example a frame rate of 25 frames per second is required, whereas only 5 to 10 frames can be provided. This example also shows that QoS properties defined by a minimum and a maximum value can be matched to properties defined by a set of options.

When a provided video streaming interface of a dock is bound to a required interface of another dock, the QoS ranges of these interfaces are merged into one new range for both interfaces. This means that both interface definition instances are changed to a range or an option that they both support. Additionally, this binding also has consequences for other video streaming interfaces that reside on any of the two docks. For example, if the QoS range of a provided interface of a dock is changed, this has consequences for a required
6.2. Interface matching

This situation is depicted in Figure 6.3 and is called Quality of Service propagation. The first step depicts three docks, their interfaces and the corresponding frame rate QoS properties. When in the second step, the video filter is connected to the video sink, the frame rate ranges of the interfaces are merged. Because the video filter dock now needs to produce a video stream with a frame rate of at least ten frames per second, it also needs to receive a video stream with a minimal frame rate of ten. If in the third step of Figure 6.3 a connection is made to the required interface of the video filter dock, the frame rate of this connection becomes fixed because only one value is possible. This value of 10 frames per seconds is propagated from the required interface of the video filter dock to the provided interface of the same dock.

If the video filter dock would output more frames than it receives, it would be sending duplicate frames which is in general a waste of bandwidth. This is not the case when the functionality of a dock is, for example, to up-sample and interpolate a given video stream. A ratio between the required and provided interfaces should be configurable. The current implementation assumes this ratio is one. Expansion of the matching algorithm to support these kind of interface dependencies is left for future work. An other extension to the current implementation is an algorithm that, given the docks that are to be connected, calculates a transitive closure.

Besides defining QoS ranges per property, it might be desirable to enable definition of dependencies between QoS parameters. If, for example, the video resolution is set to 800x600 a minimal frame rate of 25 frames per second might be required, whereas a resolution of 1024x768 might only require a frame rate of 15 frames per second because of the higher resolution of each frame that is received. Dependencies between QoS properties could be defined by using <if>, <then> and <else> elements in the QoS description. Also, whole parameter presets could be defined of which one might be chosen. A more extensive interface matching algorithm is required if this kind of functionality is to be supported. The
6.2.2 Object data Interfaces

When matching a provided object data interface the matching algorithm searches for required object data interfaces that have the same datatype. The datatype is the tag name of the only child element of the <objectData> element. Figure 6.4 depicts four interface definitions. All definitions have the same datatype, namely “carInfo”. Definitions (a) and (b) are matching because each child element in the required interface is present in the provided interface. If a child element of the required interface would have a complex type, all of the child’s children must be present in the corresponding child element of the provided interface.

This matching technique enables dock developers to easily extend existing datatypes. If for example, color information on cars is required, it can just be added to the interface description of the dock, resulting in the definition (c) of Figure 6.4. This interface definition will still match with interface (b). The required interface will just ignore the color information.

This will not work the other way around; if an additional element is added to the datatype in the required interface, and not in the provided interface, the two interfaces are no longer matching. So the addition of the <licencePlate/> element, resulting in definition (d) of Figure 6.4, makes the interface no longer matching with (a) and (c).
6.3 Interface binding

Binding an interface in the ViFramework means that the interface knows its peers and that all the information needed to create the required connections is available. The connection will not be made before the dock is started. All information needed for a connection that is to be set up is stored in the interface instance description by adding a <binding> element to the <interface> element. Provided interfaces support connections up to more than one required interface, so more than one <binding> element can be present, each with its own QoS properties. An example of a dock with one provided interface that is bound to two required interfaces is shown in Figure 6.5. A local connection is made for a target dock that resides on the same host as the sending dock and a [RTP] connection is set up for another target dock residing on a different host. Additional binding information can be added, for example the port number that is needed for the [RTP] connection is also stored here.

To check whether an interface is bound can be done using this <binding> elements.
Figure 6.5: Interface matching example: interface binding

Only the existence of a <binding> element needs to be checked. When unbinding an interface from a certain connection the <binding> element will be removed from the description again. The QoS properties of the dock may be different from the properties of the interfaces. When taking Figure 6.5 as an example again, the dock should operate at a minimum of 25 frames per second. It is the task of the interface to down-sample the video stream for the second binding that only required 15 frames per second. This can be accomplished by using an internal frame buffer which receives all frames from the dock, but from which only 15 frames per second are read and send to the target interface of the target dock.

Besides the frames per second QoS property, this ViFramework implementation supports resolution, bit rate and gop size. Future implementations could expand on this by supporting additional QoS properties (e.g. codec choice).

### 6.4 Communication principles

When two interfaces are connected to each other, the relative position of the two docks will determine what kind of connection will be set up between these interfaces. It is up to the distribution manager to find out what type of connection should be created. If both interfaces reside on the same host, a connection should be setup using local shared memory. If the two docks reside on different hosts, a network connection needs to be created.

For networked connections a distinction between video streaming and object data interfaces
If a network connection between two components spans more than one LAN (and therefore needs to traverse a NAT router or firewall) no RTP connection can be setup because this protocol is IP address based and hosts behind a NAT router do not have a unique IP address. Furthermore, firewalls could block the ports used by the protocol. A SOCKS 5 proxy is used to setup a SOCKS 5 byte stream (XEP-0065) between the two components. Such a byte stream is based on a Transmission Control Protocol (TCP) connection and is therefore not very suitable for applications subject to QoS.

Typically, the real-time part of video content analysis applications resides on a LAN whereas Wide Area Network (WAN) connections are, due to their higher delays, mostly used for monitoring, control and notifications. For the latter purposes guaranteed delivery is more important, which makes a SOCKS 5 byte stream a suitable candidate for inter-LAN connections.

![Possible network structure supported by the ViFramework](image)

Figure 6.6: Possible network structure supported by the ViFramework. End-to-end RTP sessions are used for intra-LAN video streaming. For inter-LAN video streaming the XMPP server is used as SOCKS 5 proxy. Only the XMPP server requires a public IP address. Metadata communication is not depicted in this figure.
When communicating object data over the network, the XMPP protocol will be used irrespective whether the communication needs to traverse router and firewalls or not. So object data travels back and forth between the ViFramework and the XMPP server. End-to-end connections could be used for intra-LAN communication of object data, but this requires an extension of the XMPP protocol (XEP-0246). Because the vast majority of the data communicated within a typical video content analysis application is video data, there is little to gain with respect to bandwidth. Therefore, this extension is not implemented.

When a dock is started, it sends data to an interface and it is the responsibility of that interface to send this data to all interfaces that are bound to it. For each binding element in the interface definition, a copy of the data will be send. In case of video data, additional processing might be necessary in order to provide the required QoS level. When multiple targets share the same medium and require more or less the same QoS properties, multicast constructs might save bandwidth usage. Implementation of such multicast constructs is left for future work.

### 6.4.1 XMPP connection

For sending object data the XMPP protocol is used. This requires the availability of an XMPP client to the object data interfaces. Additionally the XMPP protocol can be used to send triggers to video streaming interfaces if pull-based communication is required. Future functionality of the ViFramework could be that docks themselves negotiate QoS parameters on the fly and propose changes to each other, in contrast to the current situation where the distribution manager is responsible for this. Taking this into consideration, each dock is equipped with an XMPP client. The XMPP client comes with a set of event handlers that can be implemented. Currently, the arrival of messages containing object data or a trigger are forwarded to the event handlers of the appropriate interface.

The XMPP client of each dock will register itself to the XMPP server using its dock ID as identifier. When communicating object data the sending interface only needs to know the dock ID and the required interface identifier of the target dock. The objectData element will be filled and send to the target interface. At the target interface an event handler will be called when the data arrives.

### 6.4.2 Local connection

When a local connection needs to be set up the dock manager becomes owner of the communication “channel”. In case of video streaming the dock manager will create a buffer of an appropriate size where the provided interface can write its output to and the required interface can read from. These interfaces will receive a pointer to this buffer
from the dock manager. In order to allow simultaneous reading and writing to this buffer, ping-pong buffering is used.

The idea behind ping-pong buffering is depicted in Figure 6.7. It consists of two buffers, each large enough for one frame. One buffer is used to write a file to, the other one is used to read the frame from. Initially only a write operation can be done. When the writing operation has finished, the pointers to the two buffers are switched. Enabling writing another frame to the buffered image right away and enabling reading of the buffered frame in parallel. The required interface can poll the buffered image for a new frame. The pointers of the buffered image can not be switched if a read operation is in progress, in this case the switching is postponed.

Sending local object data works using method invocation. The dock manager will provide both docks with pointers to each other. The required interface has event handlers implemented that react on the arrival of object data over an XMPP channel. These event handlers can be directly called using the provided dock pointer and the interface index, stored in the <binding> tag.

6.4.3 RTP connection

Setting-up an RTP connection requires both interfaces to know the IP address of each other. Furthermore, both docks need to designate a port that is used for this connection. The provided interface needs to know what port number the required interface reserved for this connection. As is shown in Figure 6.1, this kind of additional information is added to the <binding> tag and therefore available after binding.

The RTP protocol is built upon the UDP protocol, making it suitable for time critical
communication, but also vulnerable to packet loss. This implementation only supports pull-based video streaming. This involves the required interfaces triggering the provided interface each time a new frame is needed. An advantage of this method is that the required interface will not receive more frames than it can handle, saving network bandwidth. A drawback is the added delay of the trigger message. For triggering, the XMPP protocol is used, in order to make sure triggers arrive.

The RTP protocol and the compression techniques used in the ViFramework are part of the ViNotion streaming library and are therefore not considered to be a part of the framework. Using appropriate compression techniques could make the streaming more robust towards packet loss. Also, the implementation of push-based video streaming would be possible, if the Real-time Streaming Protocol (RTSP) was available. Besides that, buffering more that one frame at the interface-level should be considered. This could decrease jitter, but will affect latencies. A good trade-off must be made, for which more research needs to be done. Improving the streaming constructs is left for future work.

6.4.4 SOCKS 5 connection

As is shown in Figure 6.6, a SOCKS 5 byte stream needs an intermediate SOCKS 5 proxy server to route the byte stream from one host to another. The SOCKS 5 proxy server, like the XMPP server, needs a public IP address. A SOCKS 5 proxy is typically a built-in feature of an XMPP server. Therefore, no separate SOCKS 5 proxy server has to be used. Figure 6.8 shows how a SOCKS 5 byte stream can be setup between two interfaces.

The SOCKS 5 protocol is built upon TCP facilitating reliable communication, but when packets are lost latency will increase. The interference of the proxy server makes the current setup unfeasible for intra-LAN communication because the video stream will consume twice the amount of the LAN’s bandwidth needed. An alternative approach where a dock can act as a SOCKS 5 proxy itself will solve this issue, but implementation of this mechanism is left for future work. The implementation of this mechanism, in combination with the current solution, could offer the end-user of the framework a choice between reliability and timeliness.
6.4. Communication principles

<table>
<thead>
<tr>
<th>Required interface</th>
<th>Proxy server</th>
<th>Provided interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send SSO initiation request</td>
<td>Open TCP socket</td>
<td>Acknowledge SSO connection</td>
</tr>
<tr>
<td>Request SSO connection</td>
<td>Acknowledge SSO connection</td>
<td></td>
</tr>
<tr>
<td>Request activation</td>
<td>Acknowledge activation</td>
<td></td>
</tr>
<tr>
<td>Exchange data over SSO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# "----" ... communications over XMPP
# "\"\"" ... communications over TCP
# "///" and "////" ... communications over SSO 5
# "====" ... communications over the byte stream itself

Figure 6.8: SSO 5 connection setup data flow [3]
Chapter 7

Implementation

This chapter elaborates on the implementation of the ViFramework developed in this project. Its functionality includes design time means for easy component instrumentation which will be discussed in Chapter 9. This chapter focuses on the created framework entities and the basic functionality provided by them.

7.1 Framework entities

Three framework entities have been developed that together form the active part of the ViFramework:

- A distribution manager tool that enables instantiation, deployment, configuration, starting, stopping and destroying of docks on available hosts.

- A graphical user interface gives the end-user full control of the distribution manager’s capabilities and to give a clear overview of the current system status.

- A dock manager tool that runs on all hosts that are to be used by the framework. This framework entity awaits commands from its distribution manager and returns the current host status, and the status of the docks on the host, back to it.

Besides this framework entities, thirteen dock types have been created that allow the demonstration of the framework’s basic functionality, enabling the setup of video streaming and object data connections between these docks. The docks created for the PTZ camera scenario are also part of this dock type collection.
7.2 Framework functionality

The Graphical User Interface (GUI) is created using the Model-View-Controller pattern [27]. The interface provides the end-user with a view on the current system status model which is managed by the distribution manager. This status includes information about which hosts are online and what docks can be deployed on these hosts. Figure 7.1 shows an example setup that has two available hosts. More information on the dock type and its interfaces is available using this GUI. New hosts that come online will be automatically added to this list.

The user can use the GUI to send commands to the distribution manager which will forward them to the appropriate dock managers. For example, docks can be instantiated by double-clicking a docktype on a specific host, after which the dock instantiation can be found in the “Deployed docks” list showed in Figure 7.2. In this part of the interface the current dock status and that of its interfaces is shown. The command buttons can be used to compose instantiated docks in order to form an application. Functionality to start and stop individual docks is provided, as are buttons to start and stop all deployed docks and thus the whole application. In the right hand table, dock parameters can be changed in order to manually configure the application, in this example the filename of the file to create a stream is configurable.

![Figure 7.1: GUI: Host information.](image1)

![Figure 7.2: GUI: Deployed docks.](image2)
Besides the textual overview of the system status as is depicted in Figure 7.2, the distribution manager tool also provides a graphical representation of the current system status. It shows the current application topology and relevant information of its hosts and docks. An example of such a generated image is depicted in Figure 7.3, it shows an application performing background subtraction on a video file and sending the result of this algorithm to the display. All the interfaces of the docks are connected and the docks themselves are started, indicating that this is a running application.

![Network topology diagram](image)

**Figure 7.3: GUI: Topology image.**

All actions and events that are relevant to the end-user of the framework are logged by the distribution manager and shown in the user interface. Error messages from anywhere in the application are propagated to the distribution manager and presented to the end-user as is shown in Figure 7.4. The logging screen shown the whole error trace, back to from where the error originated. The example figure shows an error generated when a dock reading a video file is started and the file path was incorrect.

![Log screen](image)

**Figure 7.4: GUI: Log screen.**

The PTZ camera scenario presented in Section 2.3.1 is implemented. The generated topology image is shown in Figure 7.5. This PTZ application is slightly different from the proposed scenario because no rule-based engine is implemented and no VCA is done on the output stream from the PTZ camera. The rule-based engine is not needed because the VCA algorithm that processes the video feed from the static camera, generated PTZ coordinates, and GUI information itself.

The “demo” host contains a static camera and a tracking dock. The “dom” host only contains the PTZ camera dock which gets PTZ coordinates from the tracking dock. Host
Dom2” instantiated a GUI dock that shows all information to the end-user. All docks have all their interfaces connected so all docks are deployed but the application has not been started yet.

Figure 7.5: GUI: Topology image of the PTZ scenario.

The GUI shows a list of all connections that are created in the current situation. Figure 7.6 shows the connection that are present in the example topology of Figure 7.5. Here command buttons are provided that allow the break down of currently established connections.

Figure 7.6: GUI: Connections in the PTZ scenario.

These GUI parts together provide the end-user with easy means to manage applications that can be composed from pre-existing docks. The dock manager tool does not need any user interaction after it has been started because it is controlled by the distribution manager. A configuration file that specifies the host name, the XMPP server address and the dock types that are available needs to be provided at startup.
Chapter 8

Evaluation

This chapter evaluates the developed ViFramework implementation, by comparing the current implementation to the requirements listed in Section 2.5. Furthermore, various metrics were calculated in order to perform a performance evaluation.

8.1 Requirement fulfillment

When comparing the requirements listed in Section 2.5 with the capabilities of the current ViFramework implementation there are a few requirements that are not met. These requirements are all non-critical.

Whereas in principle the software is platform independent there are a few minor parts of the software that will not run on Windows. An example of this is the resource extraction which uses Linux shell command to extract information from the operating system. Code parts that need to be rewritten for use under Windows are marked as such. This prevents the implementation to meet requirement GEN-03 (defined in Section 2.5).

Because extensive resource monitoring and profiling has been marked as future work, the distribution manager is unable to make decisions based on this kind of information. This means that load distribution is not yet possible and also warning the end-user when trying to deploy a dock on an overloaded dock is left for future work. Requirements DIM-[07-09] are not met because of this.

Despite these minor shortcomings, the implementation of the framework is capable of reproducing the scenarios presented in Section 2.3. The PTZ and the failover scenario are supported, whereas the expansion scenario lacks resource profiling and the Internet scenario lacks Windows support.
8.2 Performance evaluation

Data compression and streaming are considered to be an independent part of the framework. Therefore, no network usage measurements on applications, deployed on multiple hosts, were carried out. Furthermore, after application initialization, the only framework related network traffic is resource usage information. This traffic is negligible as compared to traffic generated by video streaming.

In order to evaluate the performance of the ViFramework implementation, the computational overhead and the time needed for host failure recovery were measured.

8.3 Evaluation method

An application was created that reads a video from file, applies object detection on the video and writes the resulting video to a display. The video used has a resolution of 640x480 at 15 frames per second. For framework evaluation the application was divided into three docks (read, process, write) which were all deployed on the same host. Besides the frameworked application, a standalone application was created executing the same algorithm on the same data. For comparison, the frameworked application and the standalone application were both executed on the same host.

The test host contains a quadcore Intel® Core™ i7 870 processor at 2.93 Ghz with 2 GB of RAM. The operating system used was Linux 2.6.36-26.

8.4 Framework overhead

Video content analysis algorithms are often computationally intensive, making it important that the framework overhead in terms of CPU usage is minimized. Furthermore, because of real-time requirements, the processing delay the framework introduces should also be minimal. To measure the overhead the framework imposes, a standalone application was compared to the same application in the ViFramework but deployed on only one host. While the algorithm was running, the CPU usage was measured for a period of three minutes. Two runs were made, the second run executing a more demanding version of the object detection algorithm. Each run was performed 10 times.

The results of the measurements are presented in Table 8.1. Both runs indicate a framework overhead of about 11.2%. For most target applications this overhead is considered
acceptable. This overhead can still be reduced as the current framework implementation is still a prototype.

### 8.5 Failover automation

On host failure the framework tries to re-instantiate a failing dock as quickly as possible in order to lose a minimal amount of data. Because of real-time requirements, no data is retransmitted, so the longer it takes to take over the functionality of the failing host, the more data will be lost. In this benchmark the three docks of the evaluation application were deployed on two hosts. The reading and writing docks are deployed on one host and the object detection dock is deployed on the other. The dock performing object detection is deliberately interrupted by killing its dock manager process. The time between this point and the point where the second host has taken over the dock on the failed host is measured in order to calculate data loss.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fps.</td>
<td>15</td>
</tr>
<tr>
<td>#/ Measurements</td>
<td>20</td>
</tr>
<tr>
<td>Min. time</td>
<td>467 ms</td>
</tr>
<tr>
<td>Avg. time</td>
<td>584 ms</td>
</tr>
<tr>
<td>Max. time</td>
<td>830 ms</td>
</tr>
<tr>
<td>Avg. frame loss</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 8.2: Overtake measurements

The results are presented in Table 8.2 and show that fast recovery is possible using the ViFramework. Losing slightly more that half a second of data on average is acceptable for most surveillance applications.
Chapter 9

Instrumentation Procedure
Generalization

The current ViFramework implementation assists dock programmers with the instrumentation procedure. This chapter will focus on the manual procedure. Generic parts of the procedure will be highlighted and some suggestions towards automation will be offered. Automation of the instrumentation procedure is left for future work.

9.1 Dock abstract base class

Instrumentation of a component starts with deriving a new dock class from the Dock abstract base class, thereby giving this newly created dock basic framework functionality such as binding, starting and stopping. The abstract base class also contains a pure virtual method. This controlLoop() method needs to be implemented by the dock developer and must contain the dock’s active behavior. The starting of a dock executes the controlLoop() procedure of the dock in a separate thread, starting the active functionality of the dock.

The dock can also react on arriving events in the form of object data arriving at the XMPP handler of this dock. When this happens, the handleObjectData() method will be called and the received object data will be passed as an argument. It is up to the dock programmer to reimplement the handleObjectData() method, extracting the object data from the XML tag received and taking further action.
9.2 Interfaces

Before the methods previously mentioned become meaningful, interfaces must be specified. The type and number of interfaces obviously depends on the component requirements. When, for example, a VCA dock is created, it will typically get a required video streaming interface and a provided video streaming interface. These interfaces are to be attached to the dock at instantiation, which can be done by instantiating one of the four available interface classes.

9.3 XML description

Each dock needs an XML dock description, defining its functionality and the interfaces that are used. These dock descriptions are part of the dock manager description file of which an example is listed in Appendix C. A unique dock identifier needs to be assigned to the new dock, and a description of the dock’s functionality needs to be created.

Parameters needed by the dock to configure its component can also be defined in this description. A parameter requires a name and a default value. At design time, all parameters specified in the dock description can be directly accessed by using the `getParameter()` method, providing the name of the parameter as an argument. At deployment time the framework end-user can set these parameters at the distribution manager as is depicted in Figure 7.2.

The XML description of the interfaces are discussed in detail in Chapter 6. The QoS parameters defined in this the video streaming interfaces can be accessed by the dock programmer at design time by calling the `getQosProperty()` method, providing the interface index and the name of the QoS property as arguments. At runtime these properties are changed when the interfaces are bound, and are fixed when the dock is started.

9.4 Assembly

All the aforementioned framework constructs aid the dock programmer by relaxing the effort of component instrumentation. In the current implementation of the ViFramework the dock programmer is still responsible for putting all the pieces together: Writing an XML description of the dock, instantiating appropriate interfaces, and coupling the component’s interface to the interfaces of the dock. If the docks processes video, the dock programmer must also develop dock code that waits for frames to arrive and assures that QoS properties of the dock’s interfaces are met. Converting XML based object data to a representation
usable by the component is also still a manual job.

9.5 Towards automation

Having implemented a number of docks, it has become clear large parts of the instrumentation procedure can be generalized or even automated. Although automation is not considered in this work, generic parts of the procedure will be highlighted in this section and suggestions towards automation are provided where possible.

9.5.1 Dock descriptions

As can be observed in the example dock manager description listed in Appendix C, dock descriptions are quite straightforward. This is caused by the small number of interface types the ViFramework uses. There are only two types of interface descriptions; Video streaming interfaces containing QoS parameters, and object data interfaces containing an XML representation of the data structure that is to be communicated.

A rather simple tool for dock description generation could be developed, allowing the dock developer to specify a unique identifier, a functionality description, the required parameters and the required and provided interfaces. Because there is a fixed set of QoS parameters, these can be easily set using a simple interface. The construction of an XML representation of the object data that is to be communicated could be automated for basic datatypes, but for more complex structures manual construction probably will remain necessary.

9.5.2 Code generation

Using the (generated) dock description a skeleton of the dock code could be generated. All docks need to instantiate the interfaces defined in the dock description. These interfaces need to be destroyed again when the dock is stopped.

The controlLoop() method can also be largely generated from the dock description. Instantiation of framework constructs in this loop can be generated because they depend on the interfaces that are defined. The resolution of the temporary frame used to couple an interface to the component for example, is dependent on the resolution QoS property of the interfaces.

The structure of most video streaming algorithms are the same. Typically VCA algorithms wait for a frame to arrive at the required interface, process this frame and send the results
to the provided interface. If QoS properties cannot be met or an error occurs, the dock should inform the dock manager. Code for this is generic enough to be generated by a tool.

Appendix D lists an example dock implementation performing background subtraction on a video stream. In this listing a distinction is made between framework code and component code. This example code shows that by far the largest part of the code is part of the framework. All that is framework code could be generated by a tool because it is independent of the component embedded. The dock programmer would only have to fill in the pieces of code labeled as component code in the example, i.e. component initialization and the coupling between the dock and the component.

What is not shown in the example code of Appendix D is the assurance of QoS properties. Because these properties depend on the binding of the interfaces, and not on the component code, this code can also be generated. For example, a timeout could be placed on the waiting for a new frame. This timeout would depend on the maximum fps QoS property of the provided interfaces. On timeout a warning could be raised, informing the dock manager the required frame rate cannot be met.

In contrast to docks that process a video stream, docks that process object data are not that easy to generalize. Because the focus of this thesis lies with video streaming components, this will not be discussed here.
Chapter 10

Conclusions

In this thesis the ViFramework, a framework for networked video streaming components targeted at surveillance applications, is presented. This framework provides end-users with easy to use tools that enable the deployment of distributed video streaming applications. A GUI is developed to provide the end-used with a clear view on the current system status and to give access to all framework functionality. This enables the end-user to easily create complex application architectures by configuring and composing existing docks.

For VCA developers the framework provides flexible means to specify new docks using XML as definition language. The framework provides location transparency resulting in a setting where dock developers do not have to consider how created docks are connected. Stream compression, threading and timing are also handled by the framework.

The combination with the XMPP protocol enables the framework to deploy an application on a WAN by using firewall and NAT-router traversal. This enables remote monitoring, control and notifications. Basic resource monitoring on host-level is used, and the generated information is communicated to a distribution manager in a smart and configurable manner.

The ViFramework design is largely implemented and with the created tooling the PTZ application defined in Section 2.3 was implemented. This satisfies the first project goal defined in Section 1.3 which is to deploy an existing application on multiple hosts using the ViFramework.

The instrumentation of thirteen components for usage in the ViFramework has given insights on how the instrumentation process could be generalized. Laying emphasis on video streaming components, it is argued in Chapter 9 that a large part of this instrumentation process could be automated. This realizes the second project goal, which is generalizing the instrumentation process.
The last project goal, using resource usage information to propose handling decisions, could not be completely satisfied. But, as is stated in Section 1.4, the priority of the project lies on the first two goals. When extending the resource monitoring capabilities of the framework, not much effort would be needed to also meet this last goal.

The evaluation of the framework in Chapter 8 showed that the current implementation satisfies almost all requirements defined in Section 2.5. Four low priority requirements could not be met and are listed in Table 10.1, the last three of these requirements correspond to the last project goal.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN-03</td>
<td>The software runs on Windows.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-07</td>
<td>The Distribution Manager keeps profile information of docks.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-08</td>
<td>A load distribution policy can be defined.</td>
<td>3</td>
</tr>
<tr>
<td>DIM-09</td>
<td>The Distribution Manager produces a warning and refuses to deploy when the system engineer tries to deploy a dock which the application cannot handle.</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 10.1: Unmet requirements

Measurements show that framework computation overhead is also acceptable. A failover algorithm enhances the robustness of frameworked applications. Evaluation of this algorithm shows that failover is achieved within acceptable time. Overall, the measurements of the presented prototype implementation show that the ViFramework is efficient and suitable for networked video surveillance applications.

The current design and implementation are a good starting point for future extension and some improvements are listed in Chapter 11. Some basic software metrics of the current ViFramework implementation are listed in Table 10.2. A distinction is made between the framework code, the code of the thirteen dock implementations and the GUI code. This implementation can already be used for easy distribution of video streaming applications.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Framework</th>
<th>Docks(13)</th>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of classes:</td>
<td>17</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Lines of code:</td>
<td>5824</td>
<td>1517</td>
<td>709</td>
</tr>
<tr>
<td>Lines of comment:</td>
<td>2171</td>
<td>564</td>
<td>122</td>
</tr>
</tbody>
</table>

Table 10.2: Implementation metrics
Chapter 11

Future Work

The current design and implementation of the ViFramework can act as a basis for many extensions and improvements. The created implementation is stable, easy to use and easy to extend. This chapter elaborates on improvements on the current implementation that fall within the current software design. This chapter also provides directions toward future research.

11.1 Implementation improvements/extensions

In order to improve the current ViFramework implementation, some changes to the dock manager code can be made that will decrease CPU usage overhead generated by framework. For example, an alternative architecture where all docks use the XMPP client of the dock manager instead of their own, could be considered. Parts of the current implementation can be rewritten in order to meet the platform independence requirements. This platform dependent parts are documented as such in the code documentation.

End-to-end TCP connections could be used for intra-LAN communication in order to present the end-user with the ability to choose between UDP and TCP based communication. The SOCKS 5 byte stream protocol could be used for this, as proposed in Section 6.4.4 or the RTP protocol could be used over TCP. Furthermore, the choice between pull- and push-based communication could be offered. When push based video streaming is considered, the RTSP protocol should be implemented in order to prevent the sending of more frames than can be handled.

When matching object data interfaces, the contents of the data element should be matched as well, as is proposed in Section 6.2.2. In the current implementation, only the name of the datatype is checked for matching. For matching video streaming interfaces, a
more extensive definition of the QoS properties, as is proposed in Section 6.2.1 could be implemented.

The currently implemented failover algorithm is rather simplistic. A more extensive and iterative algorithm could be implemented to make sure failover can also be performed when multiple docks where instantiated on the failing host.

11.2 Research opportunities

Using the current ViFramework design and implementation as a starting point there are many ways new research could be directed. Despite the frameworks’ constructs to easy instrument existing components, and the best practice proposed in Chapter 9, full automation of the instrumentation procedure is still far away. The first step towards this goal would be to generate dock code skeletons from the XML definitions, combined with tools that allow the easy creation of such an XML file.

An implementation of a dock repository would increase the usefulness of the ViFramework because it would allow new dock types to be uploaded to hosts at runtime. Problems here are the delays imposed by runtime compiling of dock source code and the runtime binding of these new docks to the dock manager. More research will have to be done in order to make good choices towards implementing this.

The QoS propagation construct proposed in Chapter 9 still needs to be implemented. Furthermore, extension of this algorithm needs to be investigated in order to support dependencies between required and provided interfaces of a dock. Dataflow theory will probably be helpful here. Ideally, the distribution manager will find a suitable transitive closure for the docks that need to be connected.

An application manager implementation would help the end-user to keep different applications in the framework separated. It could further relax the effort of deploying applications by allowing to pre-defined applications. An application manager implementation can use the distribution manager as interface to achieve its goals.

The framework is robust against host failure as long as the host running the distribution manager does not fail. If this happens in the current situation, the running applications are uncontrollable. This problem could be solved by adding a redundant distribution manager or by making the distribution manager fully distributed.

Additional communication styles should be explored for usage by the framework. Only point-to-point connections are supported so far, and styles like publish/subscribe and blackboard could expand the frameworks’ versatility. Research should be done finding out what communication styles have advantages in which scenarios. Also, the use of multicast
connections instead of separate point-to-point connections should be considered when a provided interface has multiple bindings.

The use of stream buffering in the interfaces should be investigated. The current implementation does not support this. Buffering will decrease jitter caused by varying network delays and packet loss, but will influence the latency imposed by the framework, making it a threat to real-time requirements. A trade-off between these two factors should be made.

Future functionality of the proposed framework should include resource usage profiling of docks on the available hosts as is done by Korostelev et. al [28]. This will enable the distribution manager to predict what resources a certain dock will use on a host. Using this information the distribution manager can deploy applications more efficiently. The dock manager could also perform resource allocation instead of the operating system which is responsible for this in the current implementation. The current resource monitoring should be reconsidered in order to extract more meaningful information. To assure a dock manager can make full use of all the host’s resources, without interference of other processes, it could be executed in a virtual machine.
References


2010.


Appendix A

Research paper
ViFramework: A framework for networked video streaming components

B. Kersten¹, K. van Rens², and R. Mak¹
¹Security and Embedded Networked Systems, Eindhoven University of Technology, The Netherlands
²ViNotion B.V. The Netherlands

Abstract—Real-time video content analysis applications for surveillance become more and more demanding. The need for load distribution, remote management and reusability calls for a component framework specialized in networked video streaming applications. Whereas lots of component frameworks exist nowadays, frameworks targeted at networked video streaming are scarce. Added requirements imposed by video surveillance applications include real-time computing and quick failover. The framework proposed in this paper meets these demands by enabling the distributed execution of video streaming applications in an efficient and resource-aware fashion. In this paper, we present the design of the proposed framework and evaluate a prototype implementation. The results of this evaluation show that this implementation is efficient and can successfully perform failover handling, making it suitable for distributing surveillance applications.

Keywords: Software component framework, video streaming, video surveillance applications, distributed video analysis

1. Introduction

Developing applications using the Component-Based Software Engineering (CBSE) paradigm [1] has many advantages such as high reusability, low time to market and decreased development costs. Many software component frameworks exist nowadays, but frameworks supporting video streaming are rare, especially when network functionality is required.

Applications that need streaming video are Video Content Analysis applications, such as the ones studied in the recent ITEA2 research projects CANTATA [2] and ViCoMo [3]. These applications are becoming increasingly more demanding. Applications that process video streams originating from multiple cameras with computationally-intensive algorithms like object detection and tracking are becoming more common. Due to the high-volume nature of video data, processing components often have high resource demands. The need for distributed applications is motivated by the need for geographical distribution and load distribution in order to make the applications more scalable.

A framework for networked video streaming components is needed, in order to enable component distribution over hosts connected by a network. The framework must enable components to be configured and composed remotely in order to form an application. By supporting dynamic reconfiguration the framework must allow for run-time modifications of the application’s component graph.

At design-time, the framework must provide component developers with abstraction of tasks, such as setting up network connections, video compression, timing and multithreading. To make the framework suitable for rapid prototyping, the framework must be flexible and component descriptions easy to adapt. At run-time, the framework should provide network-transparent means to compose an application. In order to allow applications to span both LAN’s and WAN’s, the framework must support NAT-router and firewall traversal.

Targeting at video surveillance applications, the framework is subject to real-time requirements. In general, a trade-off must be made between timeliness and guaranteed delivery. By adjusting QoS parameters the framework must be able to meet the real-time requirements of the applications. In host-failure situations the framework must be able to perform quick failover, thus increasing robustness and minimizing the amount of lost data.

This paper proposes a framework for networked video streaming components aimed at surveillance applications. An implementation of the proposed framework is presented in [4] and is evaluated in this paper. Evaluation is done by porting an existing surveillance application to the framework after which overhead and failover time is measured.

In Section 2 the general architecture of the proposed framework is presented. An application scenario is sketched in Section 3. Section 4 elaborates on framework details. Framework evaluation is presented in Section 5. Section 6 describes related work and Section 7 concludes the paper.

2. Framework Architecture

The proposed framework exists of a design-time and a run-time part. The design-time part of the proposed framework consists of means that help the component programmer to create components that comply with the framework. This includes an interface definition language, automatic code generation and programming guidelines.

Before existing video content analysis algorithms can be used as components in the framework, they are supple-
A video content analysis application consisting of four components spanning two hosts is set up. Each dock manager manages all docks on the host it resides on and one distribution manager controls the whole application.

The term dock is borrowed from the SOFA framework [5] and denotes a container for multiple components that together provide functionality to the environment. Docks include a control part which controls the included components, handles configuration and facilitates network usage, and a functional part which is the component code.

The run-time part of the framework consists of two active entities that enable the distribution of a video streaming application. Each host that takes part in the framework runs one process that manages all docks on that host. On each host, this dock manager is the only process that can instantiate docks. After instantiation the dock manager can configure, start, stop and destroy a dock. Configuration includes binding of the component’s interfaces in order to connect them to other components.

All dock managers are connected to a central service named the distribution manager that is used to gather information about available hosts from their dock managers. Since the distribution manager has control over all available dock managers, it is capable of composing a networked video streaming application, instantiating and connecting available components. A user-interface to the distribution manager enables end-users of the framework to manually setup and manage predefined applications. An overview of the high-level framework architecture is depicted in Fig. 1.

When considering video streaming applications as done in [6], three component types can be distinguished:

- **Input:** Components that capture video data from an input source (e.g. a camera, a file or an Internet stream), convert it to a common internal format, after which it can be offered to an interface.
- **Processing:** Components that can be used to read video data in the common internal format from an incoming interface, process the video stream before forwarding it to an outgoing interface in the same format.
- **Output:** Components that accept the common internal format from an interface and convert it to an output format which can, for example, be a display or a video file.

Typically, processing components can reside on any host, whereas in- and output-components need additional hardware in order to fulfill their task and are therefore located in the proximity of these devices (i.e. a camera or video display). If not composed manually, it is the distribution manager’s responsibility to setup a pre-defined application taking into account what resources are available on the connected hosts.

The distribution manager is capable of performing failover by re-instantiating failed docks on other hosts and re-routing the data through the re-instantiated docks. In the same way, the distribution manager is capable of performing load distribution. Because video streaming uses a lot of network bandwidth the framework takes network capacity into account when setting up and managing applications. It does so by adjusting stream routes and choosing appropriate Quality of Service (QoS) levels and compression techniques. By using NAT-router and firewall traversal it is possible to deploy applications that cross the borders of a LAN.

### 3. Application Scenario

As a proof of concept the proposed framework implementation is used to distribute an existing surveillance application over multiple hosts. The application chosen for this is an object-tracking application using a static and a Pan-Tilt-Zoom (PTZ)-camera as depicted in Fig. 2. The video stream from the static camera is used for object detection and tracking. When an object is detected, the PTZ-camera is used to zoom in on the target and to extract more object-specific information. The video stream from the PTZ-camera could, for example, be used for face recognition on the zoomed-in object. The PTZ-camera is controlled automatically using the coordinate information from the “Video Content Analyser” component which analyses the video feed from the static camera. Moreover, the end-user can at any time connect to
the user interface component and watch the incoming video streams and application-generated metadata. Optionally, the user can also take manual control of the PTZ-camera. This application can be divided in up to four docks:

- **Static-camera analysis**: Object detection and tracking algorithms generating PTZ-coordinates based on the video stream provided by the static camera. Metadata describing the objects and their locations is sent to a rule-based engine.
- **PTZ-camera analysis**: Controlling the PTZ camera based on incoming PTZ-coordinates and using the video stream from the PTZ-camera as input for a video content analysis algorithm.
- **User interface**: Presentation / interaction component.
- **Rule-based engine**: Gathering metadata from both analysis components and informing the end-user on events by forwarding them to the user interface.

A possible distribution of these docks is depicted in Fig. 3.

**4. Framework Details**

**4.1 Location transparency**

The ViFramework provides generic means that allow the end-users to deploy and connect docks on available host irrespective of the underlying network topology. In order to create this location transparency the framework is built on top of the XMPP protocol [7] originally designed for messaging purposes. Because of its modularity and ease of extensibility it has become a coordination protocol used by all kind of applications such as a the Peer-to-Peer desktop grid computing substrate [8]. XMPP has very attractive features for this framework such as presence information of clients, possibility of NAT-router and firewall traversal and extensive security measures like TLS ans SASL. Because of its modularity, a light-weight framework can be created by including only the XMPP modules that are necessary. All this makes the XMPP protocol an excellent network substrate for an easy to extend component-framework that satisfies the needs of demanding video content analysis applications.

A major drawback of the XMPP protocol when used for the ViFramework is the lack of efficient video streaming support. XMPP does have an extension that facilitates stream initiation (XMPP extension number XEP-0095) that is used for our own video streaming algorithms.

The main types of data that are communicated between components in a networked video streaming application are video-data, metadata and control-data. The framework supports these data types. The metadata and control data are assumed to be event-based and are communicated using the XMPP protocol itself. This protocol is XML-based and can therefore be used to send any data type that can be represented by structured text. Streaming video is done outside the protocol. The ability to communicate these data types is sufficient for the application example in Fig. 3. In general, these data types are sufficient for almost any network video streaming application.

For video streaming three types of communication are used dependent on the relative location of the components that are connected to each other:

- **Local**: For docks that are instantiated on the same host, shared memory is used for communication. The dock manager manages this shared memory.
- **RTP**: For connections between docks that reside within the same LAN the RTP protocol is used. Using RTP upon UDP makes it makes possible to meet real-time requirements because the protocol will not wait for lost packages.
- **SOCKS 5**: For connection between docks that reside on distinct LAN’s (and therefore needs to traverse a NAT-router or firewall) no RTP connection can be setup because this protocol is IP-address based and hosts behind a NAT-router do not have an unique IP-address. Furthermore, firewalls could block the ports used by the protocol. A SOCKS 5 [9] proxy is used to setup a SOCKS 5 byte-stream between the two components. Such a byte-stream is based on a TCP connection and is therefore not very suitable for applications subject to QoS.

Typically, the real-time part of video content analysis applications resides on a LAN, whereas WAN connections are, due to their higher delays, mostly used for monitoring, control and notifications. For the latter tasks guaranteed delivery is more important, which makes a SOCKS 5 byte-stream a suitable candidate for inter-LAN connections.

For metadata communication, the XMPP protocol is used, except for intra-host communication, for which we use method invocation. The message passing XMPP protocol needs an XMPP server to relay messages between hosts. End-to-End connections can be used for intra-LAN communication of metadata but this requires an extension of the XMPP protocol (XEP-0246). Because the vast majority of the data communicated within a typical video content analysis application is video data, there is little to gain and therefore, this extension is not implemented.

Fig. 4 depicts a possible network structure supported by the framework. End-to-End RTP sessions are used for intra-
LAN video streaming whereas the XMPP server can be used as a SOCKS 5 proxy in order to setup a SOCKS 5 byte-stream between two hosts in different subnets. The XMPP Server needs to be accessible from both subnets so a public IP-address is required.

4.2 Interface definition

The typical pipe and filter architecture pattern [10] found in video streaming applications consists of an input- and output component with one or more intermediate processing components. The need for dynamic reconfiguration calls for a data-centric composition technique. The proposed framework allows dock builders to specify what data types the dock requires and provides. When deploying an application the required docks are instantiated. An interface-matching algorithm is used to calculate, given an interface, which interfaces can be connected to it. The result of this algorithm can be used to automatically set up an application or can aid the user in manually setting up the application.

The demand for flexible dock definitions and the use of the XMPP protocol makes XML an appropriate language for dock and interface definitions and it is therefore used as the Interface Definition Language (IDL) in the proposed framework. At design-time, a configuration XML file is designed for each host. At start-up, this file is read by the dock manager which parses, amongst others, its identifier, the XMPP server address and the available dock definitions from this file. A dock definition contains a dock identifier, a functionality description and a list of interfaces with their respective QoS properties. For each dock instantiated by the dock manager, a copy of the dock definition is made, which can be modified by the dock manager. Changes to these description instances can be made to, for example, bind interfaces by adding target information to the interface element or to set QoS properties.

Each video streaming interface can set a topic, which can be used for stream identification and a number of QoS parameters. Two examples of interface definitions are listed in Fig.5. The matching algorithm checks whether the provided interface can meet all demands of the required interface which is the case in this figure. Metadata interfaces are defined by the XML representation of the object they communicate. The dock builder is able to construct any data type for communication as long as it is representable in XML. When, for example, the dock builder needs information about detected cars to be communicated, interface definitions as depicted in Fig. 6 can be specified. When a dock with a provided interface having this specification sends data, it fills the  element with data and sends it to the required interface. This representation allows easy creation of new object types and easy extension of existing ones.

At run-time a dock can be easily replaced by an other dock with compatible interfaces but with potentially different functionality. The user receives an overview from the framework on what connections can be made between instantiated docks. With the proposed framework, creating more complex component graphs is quite straightforward as provided interfaces are able to setup connections to multiple required interfaces. When streaming video, each of these connections can have its own QoS properties. Moreover, this allows run-time extension of existing applications by adding additional processing steps, or by branching the video stream at a certain point, in order to create a separate processing path.

4.3 Host failure recovery

The use of the XMPP protocol as a network substrate provides the proposed framework with information about the presence of dock managers. The XMPP server will notify the distribution manager when a host has gone off-line. The distribution manager will react on such an event by starting a recovery algorithm. This algorithm tries to reinstantiate the docks that were running on the failing host, on other (possibly unused) hosts, tries to reconnect them and upon success, restarts the failed part of the application. An
4.4 Resource Management

To enable automatic deployment of new applications, dynamic reconfiguration of existing application and host failure recovery the distribution manager needs information about the available resources (e.g. CPU, memory, network bandwidth) on each connected host. To enable load balancing, also information about the current resource usage is required from each connected host. Resource information is gathered by the dock managers and forwarded to the distribution manager. This enables resource management on two levels; at host-level and at system-level.

Ideally, the dock manager process is the only process running on each host apart from mandatory OS processes. Because of the low resource usage of OS processes it can be assumed the dock manager has all the host’s resources at its disposal. As future work, this could be forced by running the dock manager in a virtual machine. The dock manager will spawn a new thread for each dock it instantiates. At this point resource reservations can be made for this new dock. Docks are allowed to spawn new threads themselves. In the current implementation, host-level resource management is left to the operating system.

At system-level, the distribution manager has knowledge about the available and used resources of each host. Therefore, it can make educated decisions when deploying new docks. For example, when the new analysis dock is instantiated in the host failure recovery situation of Fig. 7, the distribution manager will opt for the unused host, rather than a host that is already doing heavy computation.

Resource requirements for video content analysis algorithms are often data-dependent [11]. This requires the framework to respond to a sudden increase in resource requirements. If a host cannot meet the resource requirements of its docks, the distribution manager needs to redistribute the application in a more appropriate way.

So, to make its global deployment decisions, we see that the distribution manager needs resource information that is as accurate and recent as possible. As stated in [12], large applications that constantly send resource information to a central service create an extensive network usage overhead. This makes an implementation in which the dock managers send resource usage updates to the distribution manager at a high fixed rate not well scalable and therefore unsuitable for a video-streaming framework. In [12] a solution for this problem is proposed. This solution divides resource usage in three usage-levels and only sends on level transitions. The ViFramework uses a similar, but more extensive solution to solve this problem.

For each resource (e.g. CPU usage) a new value will only be reported if it exceeds a user-defined threshold with respect to the last reported value and only when this situation persists for a user-definable duration. Fig. 8 shows an example resource graph. This solutions enables a trade-off to be made by the end-user of the framework between network bandwidth usage and information granularity. It is an improvement over [12] because it provides the end-user with more detail when needed and it prevents large data bursts when resource usage oscillates between two usage-levels.

5. Framework Evaluation

In order to evaluate the proposed framework, the computational overhead and the time needed for host failure recovery were measured. Because data compression and streaming, although configurable by the framework are not dependent on the framework, no network usage measurements for applications deployed on multiple hosts are carried out. Furthermore, after application initialization, the only framework related network traffic is resource usage information, which is negligible.
Table 1: Overhead Measurements

<table>
<thead>
<tr>
<th>Cpu usage</th>
<th>Standalone1</th>
<th>Frameworked1</th>
<th>Standalone2</th>
<th>Frameworked2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>58%</td>
<td>65%</td>
<td>84%</td>
<td>95%</td>
</tr>
<tr>
<td>Avg</td>
<td>60%</td>
<td>67%</td>
<td>87%</td>
<td>97%</td>
</tr>
<tr>
<td>Max</td>
<td>62%</td>
<td>70%</td>
<td>89%</td>
<td>99%</td>
</tr>
</tbody>
</table>

5.1 Evaluation method

An application was created that reads a video from file, applies object detection on the video and writes the resulting video to a display. The video used has a resolution of 640x480 at 15 frames per second. For framework evaluation the application is divided into three docks (read, process, write) which are all deployed on the same host. The frameworked application and the standalone application were both executed on the same host.

The test host contains a quadcore Intel® Core™ i7 870 processor at 2.93 Ghz with 2 GB of RAM. The operating system used is Linux 2.6.36-26.

5.2 Overhead

Video content analysis algorithms are most often computationally intensive, making it important that the framework overhead in terms of CPU usage is minimized. Furthermore, because of real-time requirements, the processing delay the framework introduces should also be minimal. To measure the overhead the framework imposes, a standalone application is compared to the same application in the proposed framework but deployed on only one host. The frameworked application and the standalone application were both executed on the same host.

The results of the measurements are presented in Table 1. Both runs indicate a framework overhead of about 12%. For most target applications this overhead is considered acceptable, and can be improved as the current framework implementation is still a rapid prototype.

Table 2: Failover Measurements

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># Measurements</td>
<td>20</td>
</tr>
<tr>
<td>Min. time</td>
<td>467 ms</td>
</tr>
<tr>
<td>Avg. time</td>
<td>584 ms</td>
</tr>
<tr>
<td>Max. time</td>
<td>830 ms</td>
</tr>
<tr>
<td>Avg. frame-loss</td>
<td>9</td>
</tr>
</tbody>
</table>

5.3 Failover automation

On host failure the framework tries to re-instantiate a failing dock as quickly as possible in order to lose a minimal amount of data. Because of real-time requirements, no data is retransmitted, so the longer it takes to take over the functionality of the failing host, the more data will be lost. In this benchmark the three docks of the evaluation application were deployed on two hosts. The reading and writing docks are deployed on one host and the object detection dock is deployed on the other. The dock performing object detection is deliberately interrupted by killing its dock manager process. The time between this point and the point where the second host has taken over the dock on the failed host is measured in order to calculate data-loss. The results are presented in Table 2 and show that fast recovery is possible using the proposed framework. Losing slightly more that half a second of data on average is acceptable for most surveillance applications.

6. Related Work

In [5] the advanced component system SOFA 2.0 is presented which was created in order to overcome limitations of formerly existing component-based systems. Due to the lack of video streaming support and the service-oriented nature of SOFA 2.0 this component system is considered unsuitable for real-time video streaming applications. Nevertheless, this work inspired some aspects of the proposed framework such as docks being instrumented components and the dynamic re-configuration of deployed applications.

In [13] the OpenDDS component framework is presented which supports complex data flows and dynamic reconfiguration. The drawbacks of OpenDDS are; the lack of video streaming support, the absence of security algorithms and problems with NAT router and firewall traversal. Another problem is the inflexibility of the framework when designing components for rapid prototyping, dynamic data types, for instance, are not supported.

In [14] the GStreamer framework is presented that focuses on audio and video streaming applications. The framework aims at creating single machine multimedia applications by composing existing components called plug-ins. The frameworks lacks presence information which is a main feature of the proposed framework and has no built-in means that support dynamic reconfiguration.
In [8] a network substrate for desktop grid computing named Orbweb is presented. This work describes the effort that is made to extend the XMPP protocol in order to meet the substrate needs. This substrate uses XMPP for NAT and firewall traversal and takes advantage of the available security protocols embedded in XMPP. Orbweb is considered unsuitable because no functionality for real-time applications is available.

In [6], Westerink proposes a flexible framework for building multi-media streaming applications. This framework identifies the general architectural structure of streaming applications and using this knowledge to create an easy-to-use framework which is used for some existing applications. The framework proposed by Westerink is targeted at creating single machine applications from existing components, and therefore not suitable to be used as a networked component framework.

7. Conclusion

In this paper the ViFramework, a framework for networked video streaming components targeted at surveillance applications, is presented. This framework provides dock component builders with flexible means to specify docks using XML as definition language. The end-users are provided with easy-to-use tools to create complex application architectures from the available docks. The combination with the XMPP protocol enables the framework to deploy an application on a WAN by using firewall and NAT-router traversal. This enables remote monitoring, control and notifications. Basic resource monitoring on host-level is performed. Gathered information is communicated to a distribution manager in a smart and configurable manner in order to enable application wide load balancing.

A failover algorithm enhances the robustness of the frameworked application. Evaluation of this algorithm shows that failover is achieved within acceptable time. Measurements show that framework has an acceptable computation overhead. Overall the measurements of the presented prototype implementation show that it is efficient and suitable for video surveillance applications.

8. Future work

Future functionality of the proposed framework will include resource usage profiling of docks on the available hosts as is done by Korostelev et. al in [15]. This will enable the distribution manager to predict what resources a certain dock will use on a host. Using this information the distribution manager can deploy applications more efficiently. The dock manager will also perform resource allocation instead of the operating system which is responsible for this in the current implementation.

The current implementation only supports end-to-end connections. More interface types are to be developed to enable other communication constructs such as publish-subscribe and multi-cast.

For now it is assumed that all docks are pre-compiled on the hosts used by the framework. A Dock Repository will be developed that allows run-time uploading of docks to hosts. This facilitates adding “blank” hosts to the system on which, on demand, appropriate docks can be installed.

Optimizations to the framework can be made in order to reduce CPU usage overhead.

Acknowledgment

The research reported in this paper has been done in the context of the first author’s master’s project. The project has been carried out at ViNotion B.V. and the support received from the company and its staff is gratefully acknowledged. Furthermore, we thank Johan Lukkien and Egbert Jaspers for their comments on an earlier version of this paper.

References

Appendix B

Dock Manager XML Schema

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- definition of simple host elements -->
  <xs:element name="name" type="xs:string"/>
  <xs:element name="server" type="xs:string"/>
  <xs:element name="startPortRange" type="xs:integer"/>

  <!-- definition of simple dock elements -->
  <xs:element name="id" type="xs:integer"/>
  <xs:element name="functionality" type="xs:string"/>
  <xs:element name="default"/>

  <!-- definition of complex host elements -->
  <xs:element name="hostInformation">
    <xs:complexType>
      <xs:all>
        <xs:element ref="name"/>
        <xs:element ref="server"/>
        <xs:element ref="startPortRange"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

  <!-- definition of complex interface elements -->
  <xs:complexType name="qosType">
    <xs:choice>
      <xs:element name="min" minOccurs="0"/>
      <xs:element name="max" minOccurs="0"/>
      <xs:element name="current" minOccurs="0"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```
```xml
<xs:complexType>
    <xs:element name="providedInterfaces">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="interface" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:complexType>

<xs:element name="parameter">
    <xs:complexType>
        <xs:attribute name="name" use="required"/>
        <xs:attribute name="default"/>
    </xs:complexType>
</xs:element>

<xs:element name="parameters">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="dockType">
    <xs:complexType>
        <xs:all>
            <xs:element ref="id"/>
            <xs:element ref="functionality"/>
            <xs:element ref="parameters" minOccurs="0"/>  
            <xs:element ref="requiredInterfaces"/>
            <xs:element ref="providedInterfaces"/>
        </xs:all>
    </xs:complexType>
</xs:element>

<xs:element name="availableDockTypes">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="dockType" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="dockManager">
    <xs:complexType>
        <xs:sequence>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
<!-- definition of complex dock elements -->
<xs:element name="parameter">
    <xs:complexType>
        <xs:attribute name="name" use="required"/>
        <xs:attribute name="default"/>
    </xs:complexType>
</xs:element>

<xs:element name="parameters">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="dockType">
    <xs:complexType>
        <xs:all>
            <xs:element ref="id"/>
            <xs:element ref="functionality"/>
            <xs:element ref="parameters" minOccurs="0"/>  
            <xs:element ref="requiredInterfaces"/>
            <xs:element ref="providedInterfaces"/>
        </xs:all>
    </xs:complexType>
</xs:element>

<xs:element name="availableDockTypes">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="dockType" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- definition of Dock Manager -->
<xs:element name="dockManager">
    <xs:complexType>
        <xs:sequence>
```
Listing B.1: XML schema of a DockManager
Appendix C

Example Dock Manager XML Description

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dockManager xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="domXmlSchema.xsd">
  <hostInformation>
    <name>dom</name>
    <server>lan.vinotion.nl</server>
    <startPortRange>10000</startPortRange>
  </hostInformation>
  <availableDockTypes>
    <dockType>
      <id>0</id>
      <functionality>Read video file</functionality>
      <parameters>
        <parameter name="FileName" default="/snow.mp4"/>
      </parameters>
    </dockType>
    <requiredInterfaces/>
    <providedInterfaces>
      <interface>
        <videoStream topic="static">
          <qos>
            <fps>
              <min>25</min>
              <max>25</max>
              <current></current>
            </fps>
            <resolution>
              <min>640x480</min>
            </resolution>
          </qos>
        </videoStream>
      </interface>
    </providedInterfaces>
  </availableDockTypes>
</dockManager>
```
<max>640x480</max>
<current/>
</resolution>
<bitrate>
<min>1000000</min>
<max>10000000</max>
<current/>
</bitrate>
</videoStream>
</interface>
</providedInterfaces>
</dockType>
<dockType>
<id>1</id>
<functionality>Display video</functionality>
</dockType>
<parameters/>

<requiredInterfaces>
@interface>
<videoStream>
<qos>
<fps>
<min>1</min>
<max>25</max>
<current/>
</fps>
<resolution>
<option>640x480</option>
<option>800x600</option>
<option>1600x1200</option>
<current/>
</resolution>
<bitrate>
<min>10000000</min>
<max>10000000</max>
<current/>
</bitrate>
<gop>
<min>1</min>
<max>12</max>
<current/>
</gop>
</qos>
</videoStream>
</interface>
</providedInterfaces>
<dockType>
  <id>2</id>
  <functionality>Provides PTZ coordinates</functionality>
  <parameters/>
  <requiredInterfaces/>
  <providedInterfaces>
    <interface>
      <objectData>
        <ptzCoordinates>
          <p></p>
          <t></t>
          <z></z>
        </ptzCoordinates>
      </objectData>
    </interface>
  </providedInterfaces>
</dockType>

<dockType>
  <id>3</id>
  <functionality>Prints PTZ coordinates to std. output</functionality>
  <parameters/>
  <requiredInterfaces>
    <interface>
      <objectData>
        <ptzCoordinates>
          <p></p>
          <t></t>
          <z></z>
        </ptzCoordinates>
      </objectData>
    </interface>
  </requiredInterfaces>
  <providedInterfaces/>
</dockType>

<dockType>
  <id>11</id>
  <functionality>Get video-stream from PTZ-Camera</functionality>
  <parameters/>
  <requiredInterfaces>
    <interface>
Listing C.1: Example XML description of a Dock Manager
Appendix D

Example Dock Implementation

```cpp
#include "DockABC.hpp"
#include "ReceiveVideoStreamInterface.hpp"
#include "SendVideoStreamInterface.hpp"
#include <BackgroundSubtraction/BackgroundModelGMM.hpp>

namespace Vi {

/**
 * This Dock performs background subtraction on an incoming
 * video stream and outputs the background mask.
 */

class DockBackgroundSubstraction : public Dock {

public:
    DockBackgroundSubstraction(unsigned int dockId);

    /// Dock functionality, reimplemented from Dock.
    void controlLoop();

private:
    /// Provided video streaming interface.
    Vi::SendVideoStreamInterface* proInterface0;

    /// Required video streaming interface.
    Vi::ReceiveVideoStreamInterface* reqInterface0;
};
}
```

Listing D.1: Header of a dock implementation performing background subtraction.
#include "DockBackgroundSubstraction.hpp"

Vi::DockBackgroundSubstraction
::DockBackgroundSubstraction(unsigned int dockId)
  : Dock(dockId)
{
  // ------- Framework instantiation:------------
  proInterface0 = new Vi::SendVideoStreamInterface();
  proInterface0->dockId = dockId;

  reqInterface0 = new Vi::ReceiveVideoStreamInterface();
  reqInterface0->dockId = dockId;

  providedInterfaces.push_back(proInterface0);
  requiredInterfaces.push_back(reqInterface0);
}

void Vi::DockBackgroundSubstraction::controlLoop()
{
  try
  {
    // ------- Framework initialization:-----------
    SendVideoStreamInterface *proInterface0 =
        static_cast<SendVideoStreamInterface*>(providedInterfaces[0]);
    ReceiveVideoStreamInterface *reqInterface0 =
        static_cast<ReceiveVideoStreamInterface*>(requiredInterfaces[0]);

    // Couple interface to xmpp client
    reqInterface0->mClient = mClient;
    proInterface0->mClient = mClient;

    unsigned int frameWidth = reqInterface0->frameWidth;
    unsigned int frameHeight = reqInterface0->frameHeight;

    Vi::Timer processTimer;
    Vi::Image<> currentFrame;
    currentFrame.size(frameWidth, frameHeight);
    currentFrame.clear();

    // ------- Component initialization:-----------

    // The background processing object
    Vi::BackgroundModelGMM bgModelGMM(frameWidth, frameHeight);

    // Create the object mask image
    Vi::Image<> objectMask;
    objectMask.size(frameWidth, frameHeight);
    objectMask.clear();
}
// ------- Framework initialization:-----------

// Start interfaces
reqInterface0->start();
proInterface0->start();

// When all interfaces are started the dock is started.
status="started";

while(!boost::this_thread::interruption_requested())
{
    // ------- Framework code:---------
    while (!reqInterface0->imageBuffer->newData
        && !boost::this_thread::interruption_requested())
    {
        boost::posix_time::microsec workTime(1000);
        boost::this_thread::sleep(workTime);
    }
    // Read from image buffer
    reqInterface0->imageBuffer->read(currentFrame);
    processTimer.start();

    // ------- Component code:----------
    bgModelGMM.process(currentFrame, objectMask);
    bgModelGMM.buildBackgroundImage();
    proInterface0->send(bgModelGMM.mBackgroundMaskYUV);

    // ------- Framework code:-----------
    processTimer.get("Dock BackGroundSubstraction");
    proInterface0->stop();
    reqInterface0->stop();
    status = "deployed";
}
catch (const boost::thread_interrupted &e)
{
    proInterface0->stop();
    reqInterface0->stop();
    status = "deployed";
}
catch (std::exception& e)
{
    // Some error occurred (e.g. end of video file),
    // so stop this dock and report to the DockManager
    reportErrorToDom(id, "deployed",
                     "[DockBackgroundSubstraction]" + (std::string) e.what());
    providedInterfaces[0]->stop();
Listing D.2: Implementation of a dock performing background subtraction.
Appendix E

Matching algorithm

```java
matchInterfaces(ProvidedInterface proInterface,
                RequiredInterface reqInterface)
{
    if (reqInterface.status != "instantiated")
    {
        return false;
    }

    if (proInterface.type == "videoStream")
    {
        if (reqInterface.type != "videoStream")
        {
            return false;
        }
        else
        {
            if (reqInterface.hasAttribute("topic") )
            {
                if (reqInterface.findAttribute("topic")
                    != proInterface.findAttribute("topic")
                )
                {
                    return false;
                }
            }
            if (compatibleQos(proInterface, reqInterface))
            {
                return true;
            }
        }
    }
    else if(proInterface.type == "objectData")
    {
        if (!reqInterface.type != "objectData")
        {
            if (!reqInterface.type != "objectData")
```
```java
    return false;
```