

**The working of the SEESCOA common
test case**

*Peter Rigole
Yolande Berbers*

Report CW354, January 2003

n

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

The working of the SEESCOA common test case

*Peter Rigole
Yolande Berbers*

Report CW 354, January 2003

Department of Computer Science, K.U.Leuven

Abstract

This report describes the key points in the implementation of the common test case of the SEESCOA¹ project: a camera surveillance system. SEESCOA stands for *Software Engineering for Embedded Systems using a Component Oriented Approach*. The additions that have been made to the initial design of the surveillance application are highlighted in this report. The main design change consists in the use of a controller for loading the initial configuration.

Keywords : embedded systems, camera surveillance case, software design.
CR Subject Classification : D2.2, D2.3, D2.4



STWW-programma

SEESCOA:

Software Engineering for Embedded Systems
using a Component-Oriented Approach

The working of the SEESCOA common test case

Peter Rigole

Yolande Berbers

{peter.rigole, yolande.berbers}@cs.kuleuven.ac.be

KULeuven, Department of Computer Science

Celestijnenlaan 200A, B-3001 Leuven

Belgium

January 2003



Abstract

This report describes the key points in the implementation of the common test case of the SEESCOA¹ project: a camera surveillance system. SEESCOA stands for Software Engineering for Embedded Systems using a Component Oriented Approach. The additions that have been made to the initial design of the surveillance application are highlighted in this report. The main design change consists in the use of a controller for loading the initial configuration.

¹The SEESCOA project is partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders)

1. Introduction

The design of the common test case got shape in a working implementation as a result of the cooperative efforts of the four collaborating universities. The main part of the implementation matches the original design, although some aberrations can be noticed which were mainly introduced for technical or practical reasons. The most important deviation is the deployment of dynamically loaded components, which added more flexibility to the working case. Other changes came forward from the distributed nature of the final case, which was not emphasized yet in the first design.

2. The controller

One of the new components that were added to the design of the test case was the Controller component. This central component has two major functions. The first one is bootstrapping components on a certain host so that the initial components get loaded as soon as the component system is started. The second function is to set up TCP/IP-connections between remote component systems in order to let them connect some of their component's ports over the remote link. The controller keeps track of all remote connections between ports, so that it can send messages over the TCP/IP-link to the right host. This way, it is the controller that makes the system work in a distributed environment.

The old fashioned way to start the component system and load a component looked like:

```
java component.system.ComponentSystem \  
component:component.examples.httpd.Httpd:HttpDaemon
```

It seems obvious that loading and linking a large group of components this way would be hard to manage. That is why the new controller is instantiated using an initialization file describing which components should be loaded and linked at startup. The following startup line, as a consequence, is not more complex than:

```
java component.system.ComponentSystem\  
component:testcases.scss.Controller:Controller:config.ini
```

The initialization file has a simple variant for using components on the local system only and it has a more elaborate version that should be used for distributing and linking components over remote hosts. We will first discuss the simple version.

The structure of the simple version of the configuration file must look as follows:

```

COMPONENT <blueprint> <componentname>
COMPONENT <blueprint> <componentname>
CONNECT <componentname/portname> <componentname/portname>
CONNECT <componentname/portname> <componentname/portname>

```

The COMPONENT command (mind the capitals!) and the CONNECT command (capitals) are the two commands. The former one has two parameters, one for the blueprint and a second for the name of the component instance. The blueprint is the complete java class name of the component. The latter command defines a connection between the ports of two components. Its two parameters consist twice of a component name attached with a slash ('/') to the port name of the ports that are to be connected. Beware that the component name can only be used in a CONNECT statement if it was defined on a previous COMPONENT line. A simple example is given below:

```

COMPONENT testcases.scss.Delay Delay
COMPONENT testcases.scss.Producer Producer
COMPONENT testcases.scss.Consumer C
CONNECT Producer/dataout Delay/datain
CONNECT Delay/dataout C/datain

```

The slightly more complex version of the configuration file managing component creation and linking on different hosts has some additional features in its configuration structure. In order to understand the configuration file, one should first understand the hierarchy and the cooperation between the controllers on the different hosts. The connections each controller sets up are immediately peer to peer thanks to a simple hierarchy between the controllers. There is one master controller and all other controllers behave as a slave controller. This means that every slave controller, as soon as it is started, connects to the master. The master then decides, based on the configuration file, which other connections (between slave controllers for example) should be set up. A slave controller automatically behaves as a local-only system when there is no master controller available.

The master controller should be defined in the configuration file with the CONTROLLER command as follows:

```

CONTROLLER <machinename> <ip:port>

```

This command has two parameters. The first, the machine name, is just a name given to the host for use in the config file. The second parameter contains the IP number of the host linked with a colon (':') to the port on which the controller listens for incoming connections. The master controller will use this command to set up the initial listening port, while the slave controllers use the command to set up the connection with their master.

The configuration file of the distributed version is also changed at another aspect. The name of each component is extended with the machine name on which it should run. This way, each controller knows which components it must load and the master controller knows which connections must be set up to comply to the requirements of the given CONNECT commands. The structure of the COMPONENT and CONNECT command now looks as follows:

```
COMPONENT <blueprint> <machinename:componentname>
COMPONENT <blueprint> <machinename:componentname>
CONNECT <machinename:componentname/portname> \
    <machinename:componentname/portname>
CONNECT <machinename:componentname/portname> \
    <machinename:componentname/portname>
```

Or in a complete example where machine2 holds the master controller having IP number 134.184.43 waiting for connections on port 2039 and where machine1 holds a slave controller:

```
CONTROLLER machine2 134.184.43.95:2039
COMPONENT testcases.scss.Delay machine1:Delay
COMPONENT testcases.scss.Producer machine2:Producer
COMPONENT testcases.scss.Consumer machine3:C
CONNECT machine2:Producer/dataout machine1:Delay/datain
CONNECT machine1:Delay/dataout machine3:C/datain
```

Furthermore, it is absolutely forbidden to add spaces at the end of the component names and tabs are taboo as well. Otherwise, the current implementation of the controller might fail to instantiate one or more components or connections.

In order to achieve consistency between the controllers, each one should be using the exact same initialization file (.ini file). To differentiate, however, between the different hosts, the machine name is

added to the startup command. In the assumption the previous example was saved in a file named `producer.ini`, the single startup line on `machine1` then looks like (mind the @):

```
java component.system.ComponentSystem \
  component:testcases.scss.Controller:Controller:\
  producer.ini@machine1
```

There is one exception in the structure of the configuration file regarding the use of the `CONNECT` command with a controller port as one of the parameters. The local controller is always named “*Controller*” and should never be preceded with its machine name. The controller has, however, only one relevant port: its “*controller*” port. This port can be accessed for dynamically loading and connecting new components. The signature of the messages allowed through this port are:

```
in CreateConnection(<String|Id1>,<String|Id2>)
in CreateComponent(<String|Blueprint>,<String|Instance>)
out ComponentCreated(<String|Blueprint>,<String|Instance>)
```

Dynamically creating a new component is now as simple as sending the following message to the local controller:

```
CreateComponent(
  <Blueprint:"testcases.scss.Decoder">,
  <Instance:"machine2:DelayBis">);
CreateConnection(
  <Id1: machine2:DelayBis/delay>,
  <Id2: machine1:Delay/delaycontrol>);
```

3. The camera surveillance case

In this chapter, we will discuss the implemented components that compose the camera surveillance system. In the first section...

3.1. Camera related components

There are two important components regarding the capturing of images and delivering them to other interested components. The Camera component is the obvious one, delivering raw images from the camera. These images are not sent immediately to the interested parties. Instead, they are first of all sent through a `VideoStreamDecoder` component which converts the raw images to any kind of requested format before sending them on to components that expect the images in the delivered format.

Because of the tight coupling between a Camera component and a `VideoStreamDecoder` component, a dynamic approach has been chosen to instantiate a decoder. The factory design pattern has been used to instantiate a certain decoder on request, based on the requirements of the requester. So, unlike the static solution, an extra port has been added to the Camera component for requesting a decoder at runtime. This `VideoStreamDecoderFactory` multiport should be connected to each component that wants to receive video images from the camera. The message sent over it is `GetDecoder` and has one parameter: the format it expects the images to have. It looks like the following:

```
VideoStreamDecoderFactory..GetDecoder( \  
    <VideoFormat: "320x240 BW,320,240,1,0">);
```

Where the string `320x240 BW,320,240,1,0` means black and white one bit per pixel images having a resolution of 320 by 240 pixels.

In return of this message, the camera sends a reply message *Decoder* holding the parameters *VideoStreamDecoder* and *VideoStreamDecoderName*. The former is a byte array containing the bytecode for the decoder and the latter holds its class name. These parameters should be used to instantiate the decoder by sending the *CreateComponent* message to the local controller, as described in the previous chapter. Once the decoder is instantiated, the factory connection may be removed. Figure 3.1 shows this runtime creation of a videostream decoder between a camera component and a component requesting video input. The functionality of the other ports of our camera component will be discussed later on in this report.

3.2. Motion detection

The motion detection component is the one that rings the alarm bell in the surveillance system. As soon as motion is detected, it makes sure the images from the camera are safely stored for later watching. The important ports of the motion detector are the *VideoStreamDecoderFactory* port, the *StreamInput* port and the *SwitchTrigger* port. The *VideoStreamDecoderFactory* port is used to request an appropriate decoder from the camera, which then sends its produced images to the *StreamInput* port of the decoder. Note that the motion detector needs to have a connection to the local controller. The *Controller* port, however, is always implicitly available in each component.

The only thing the detector does is analyzing the images for detecting motion in it, filtering out the natural noise from the camera that blurs the image. Although the computing speed of this component was thought to be a possible bottleneck for the overall speed of the system, we managed to tune the algorithm so that it needed only about thirty milliseconds to process one image while still reliably detecting any motion in the image stream. Detected motion triggers a switch component that relays the videostream to a storage controller component. Furthermore, the detector can be tuned through its *MDSettings* port with properties such as the number of frames per second it has to analyse and the sensitivity of the detection.

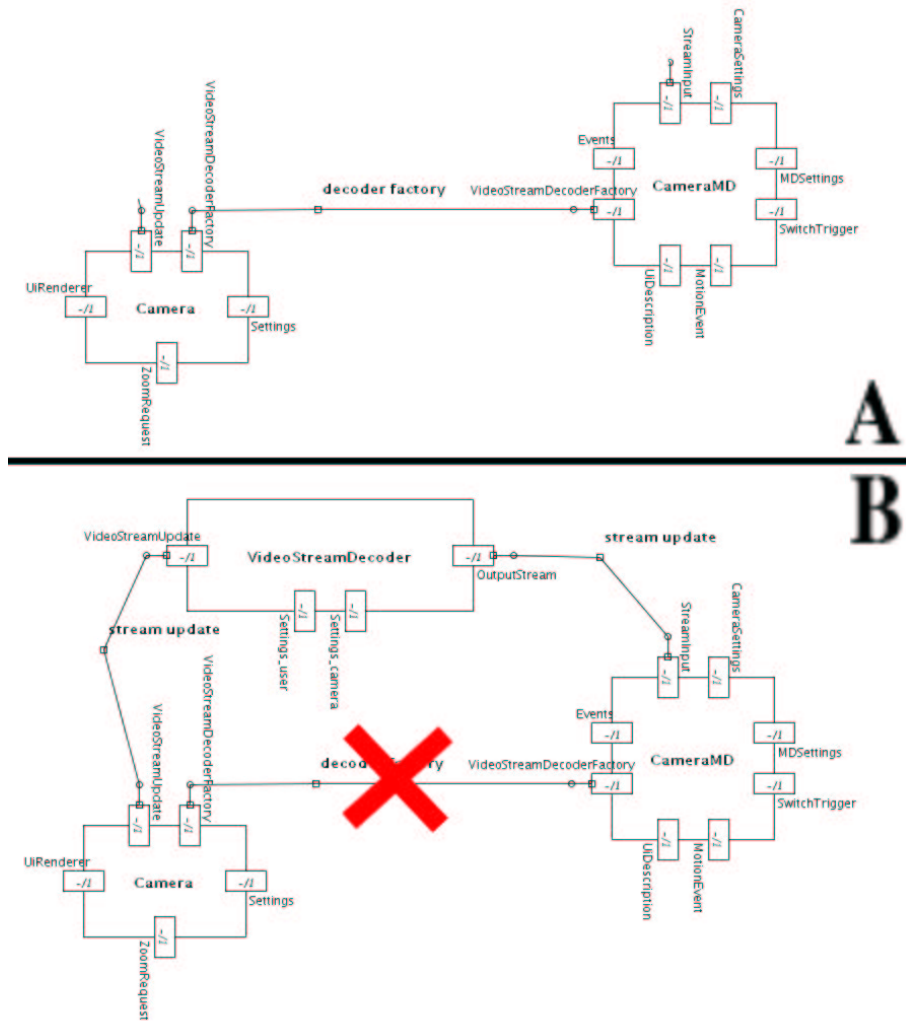


Figure 3.1: Dynamically instantiating the decoder via the factory port

3.3. Zoom behaviour

One of the goals of the case was setting up a configuration with two cameras that have a common scope of the area in their wide-angle view. In order to keep a full view on the whole area, the cameras should never zoom in on a spot at the same time. That is why a zoom behaviour component was introduced, which decided if a camera was allowed to zoom or not. If one of the two cameras is actually zooming in, then the other one should be zooming out to an appropriate view of the common area.

3.4. Storage controller

The storage controller has the important task of storing images that are considered to register objects that have caused suspicious motion events. Although the images are just filed to disk, it stores also information about the camera name and the time of occurrence indexed into a database to keep a full account of the evidence. The database makes it easier for later querying.

3.5. Rendering graphical interfaces

Some of the components in this case are able to accept input from a graphical user interface, so that the user can interact with the component from behind a computer (such as a desktop PC or a PDA¹) that can render the user interface. Such components need to offer a *UiDescription* port which can be connected to a graphical user interface (GUI) client. This GUI client serves as a proxy between the user interface and the component, so that communication is possible both ways. The rendering of the user interface itself is done by a *UiRenderer* component based on an XML description of the interface, which is stored on a location known to the component only. So, the sequence is: the component sends the description of its user interface through the *UiDescription* port to the GUI client, which in turn lets the *UiRenderer* component render the user interface. All actions between the component and its interface are handled by the

¹Personal Digital Assistant, or handheld computer

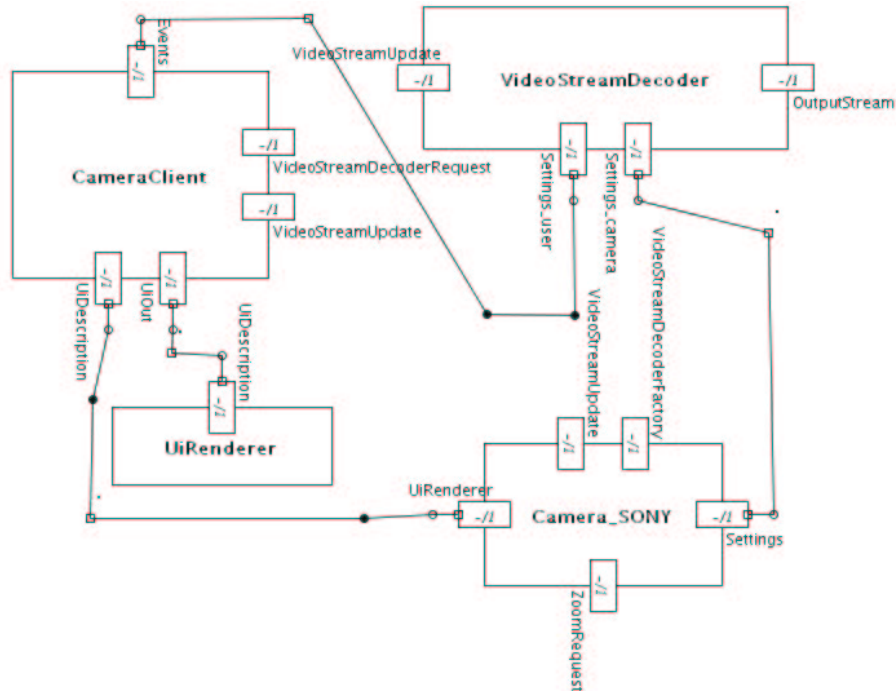


Figure 3.2: Rendering and proxying the GUI

GUI client. An example from our case is given in figure 3.2. This example has an extra indirection between the component (*Camera_SONY*) and its client (*CameraClient*) via the decoder. The *Settings* port and the *Event* port are used to guide the user interface related messages to their destination.

There are three components in our case that have an extensive graphical user interface: the camera component, the motion detector component and the storage controller component. In our case, the user interfaces of the camera and the motion detector are merged into one, as figure 3.3 shows. At the time the image was taken, no motion was being detected. Soon afterwards, however, a flow of images was received at the storage controller. A quick query on its user interface let the storage controller see what had caused the motion (see figure 3.4). The camera's user interface allows the user to zoom and to focus the camera using two slider bars (see at the left-hand side of figure 3.3). The motion detector's user interface on the other hand, enables the user to adjust its behaviour regarding its framerate, sensitivity, evolvability and logging properties. On the storage controller's user interface, many slider



Figure 3.3: GUI of the camera (left side) and the motion detector (right side)

bars are available to input the time of a possible alarm. Filling in the *CameraID* field allows one to select a camera and the *Query Image* button flushes the query data into the database, resulting either in a *No Image Found* return message or in the stored image matching the timestamp appearing on the top side of the user interface.

3.6. Distributing the case

The components in our camera surveillance case hosted on three different machines which are connected to each other with a TCP/IP network. We call the three hosts mini, tiny and trinity to distinguish them in this text. Mini and tiny are small embedded computers with a processor working at 233 Mhz, having 32 MB RAM memory and a 16 MB flash disk holding the operating system², the component system and our test case components. They are also both equipped with a Firewire card and a digital camera. Trinity, on the other hand, is a common desktop PC furnished with more RAM, CPU-power and disk space than is needed for this case. Trinity will render the graphical interfaces and store the images and database input to disk.

²a Linux operating system

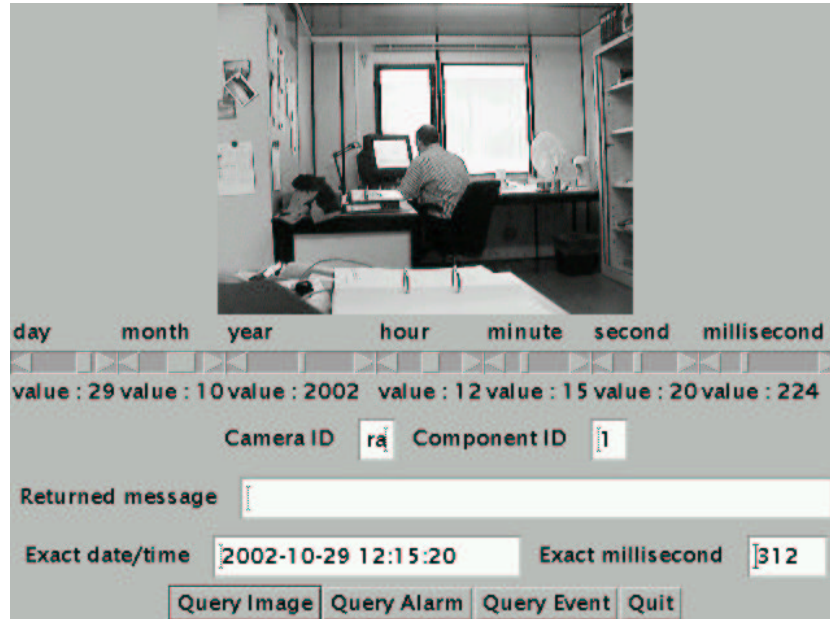


Figure 3.4: GUI of the storage controller

Mini's components

Mini is one of the two embedded machines with a camera attached to surveil the environment. The component system on mini thus needs to have at least a camera component to capture images from the camera. Other component instances on mini in our case are a motion detector, a switch component, a zoom behaviour component and a videostream decoder for each consumer of the camera's video images. Figure 3.5 gives an overview of the component structure on mini. It is important to notice some connections going loose at the right side of the figure. These are the connections that are remotely linked to other hosts (the name of the other host is written before the → sign on top of the connection). The lines in the configuration file of our case that are relevant for mini are:

```
CONTROLLER trinity 134.58.39.220:2039
```

```
##### Components: #####
```

```
COMPONENT testcases.scss.Camera_SONY \  
    mini:camera
```

```
COMPONENT testcases.scss.motiondetection.CameraMD \  
    mini:motiondetection
```

```

        mini:cameramd
COMPONENT testcases.scss.ZoomBehaviour \
        mini:zoomBehaviourA
COMPONENT testcases.scss.Switch \
        mini:switch

#***** Connections: *****

CONNECT mini:cameramd/VideoStreamDecoderFactory \
        mini:camera/VideoStreamDecoderFactory
CONNECT mini:cameramd/Controller \
        Controller/controller

CONNECT trinity:mdclient/UiDescriptionMD \
        mini:cameramd/UiDescription
CONNECT mini:cameramd/Events \
        trinity:mdclient/EventsMD

CONNECT trinity:mdclient/UiDescription \
        mini:camera/UiRenderer
CONNECT trinity:mdclient/VideoStreamDecoderFactory \
        mini:camera/VideoStreamDecoderFactory

CONNECT mini:zoomBehaviourA/camera \
        mini:camera/Settings
CONNECT mini:zoomBehaviourA/zoomRequest \
        mini:camera/ZoomRequest

CONNECT mini:zoomBehaviourA/zoomChange \
        tiny:zoomBehaviourB/zoomEvent
CONNECT mini:zoomBehaviourA/zoomEvent \
        tiny:zoomBehaviourB/zoomChange

CONNECT mini:switch/VideoStreamDecoderFactory \
        mini:camera/VideoStreamDecoderFactory

CONNECT mini:switch/Controller \
        Controller/controller

CONNECT mini:switch/Control \
        mini:cameramd/SwitchTrigger

```

```
CONNECT mini:switch/OutputStream \
        trinity:storagecontroller/VideoRecordIn
```

Tiny's components

Since mini and tiny are identical, tiny is the perfect mirror image of mini, having the exact same configuration. In a preliminary version of our case, however, the motion detector component (and thus also the switch component) is left out on tiny.

Trinity's components

The main function of trinity's components is to render the graphical interfaces of other components (such as the camera components on mini and tiny) and to manage the storage of video images containing motion events. Therefore, it has some user interface clients and renderers according to the number of user interfaces to generate of the various components (which may be scattered among the different hosts). In addition to these user interface related components, trinity also hosts the storage related components. These are the *StorageController* and the *Storage* component. The *StorageController* component offers a query interface to the database's data and relays new images containing motion to the *Storage* component. The *Storage* component then does the actual storing and retrieving of the data to and from the database. Figure 3.6 shows a visualization of the components on trinity. The GUI related components for the components on tiny, however, are left out since they are similar to those of mini. The lines of the case's configuration file concerning the host trinity are listed below:

```
CONTROLLER trinity 134.58.39.220:2039
```

```
***** Components: *****
```

```
COMPONENT testcases.scss.MDClient trinity:mdclient
COMPONENT testcases.scss.UiRenderer trinity:UiR1
```

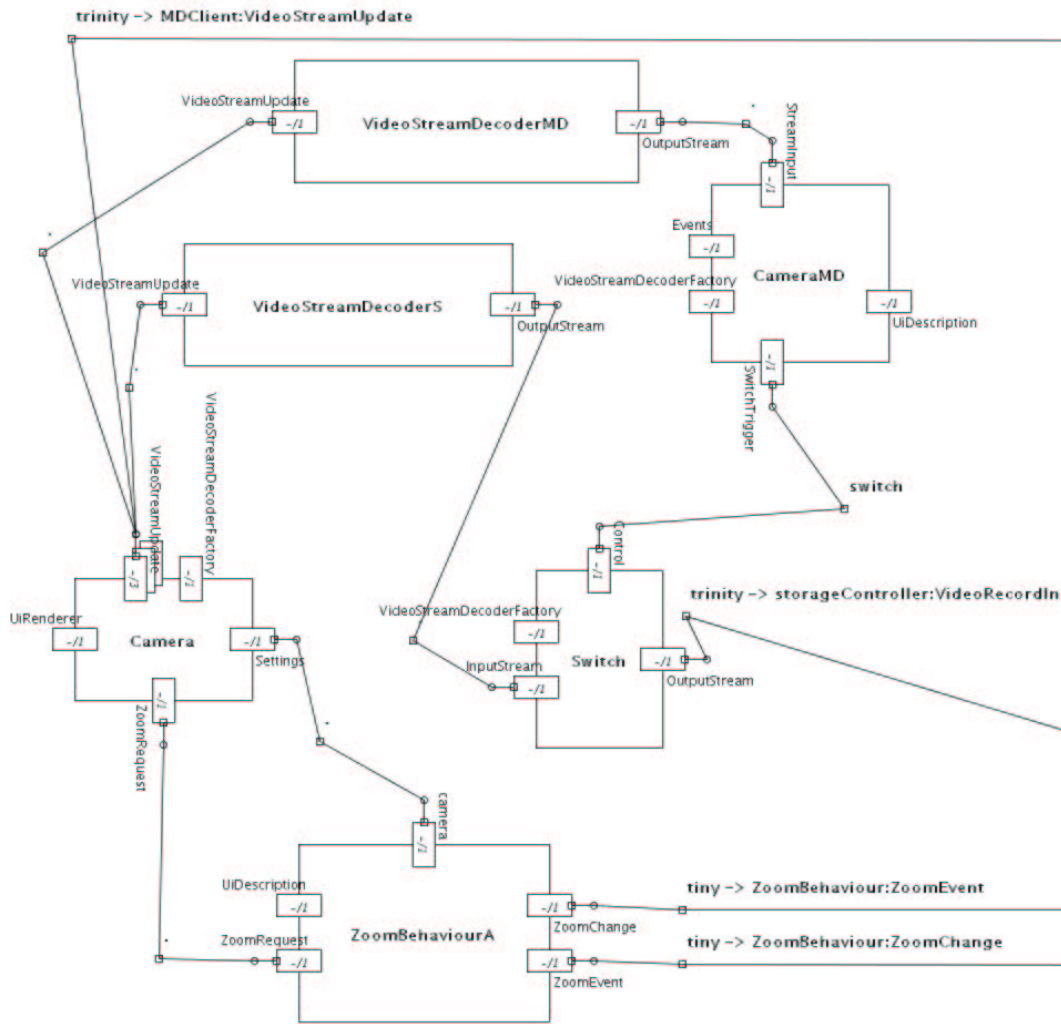


Figure 3.5: Component structure on mini

```

COMPONENT testcases.scss.TestViewer1 trinity:viewer

COMPONENT testcases.scss.storage.StorageController \
    trinity:storagecontroller
COMPONENT testcases.scss.storage.Storage \
    trinity:storage

COMPONENT testcases.scss.StorageClient trinity:storageClient
COMPONENT testcases.scss.UiRenderer \
    trinity:storageUirenderer

#***** Connections: *****

CONNECT trinity:mdclient/UiDescriptionMD \
    mini:cameramd/UiDescription
CONNECT mini:cameramd/Events \
    trinity:mdclient/EventsMD

CONNECT trinity:mdclient/UiDescription \
    mini:camera/UiRenderer
CONNECT trinity:mdclient/VideoStreamDecoderFactory \
    mini:camera/VideoStreamDecoderFactory

CONNECT trinity:UiR1/UiDescription trinity:mdclient/UiOut

CONNECT tiny:camera/VideoStreamDecoderFactory \
    trinity:viewer/VideoStreamDecoderFactory

CONNECT mini:switch/OutputStream \
    trinity:storagecontroller/VideoRecordIn

CONNECT trinity:storagecontroller/DbOut \
    trinity:storage/InOut
CONNECT trinity:storageClient/UiOut \
    trinity:storageUirenderer/UiDescription
CONNECT trinity:storageClient/QueryStorage \
    trinity:storagecontroller/QueryIn
CONNECT trinity:storagecontroller/UiDescription \
    trinity:storageClient/UiDescription
CONNECT trinity:storagecontroller/VideoOut \
    trinity:storageClient/VideoIn

```

```
CONNECT trinity:storagecontroller/DataOut \  
        trinity:storageClient/DataIn
```

Issues on distributed messages

There are many messages being send between the components in our component system and some of them flow between components on remote hosts. Since we are using a reliable TCP/IP connection between the systems, few problems should arise from this communication. We may, however, have to cope with congestion on the network since we are sending a huge amount of data through our videostreams. And congestion was, indeed, the first major problem we had in our testing stage. The problem of congestion is simply caused when a producer is producing data a lot faster than the consumer (in this case the network) can consume.

The best solution to this problem is building a Quality Of Service (QOS) assurance entity in the component system, which divides the available bandwidth amongst the different connections in accordance with the bandwidth needs of the components. At the time of this writing, however, we only implemented simple contract generating components. These contract generating components automatically insert a flow control mechanism between bandwidth-demanding connections. Their algorithm is simple: as soon as a component produces more messages (such as video images) than the network can drain, some number of messages are dropped according to the size of the congestion. It is obvious that dropping messages can not be applied in any situation. Many messages must be delivered or it may cause the application to fail when the handshaking between two ports gets corrupted.

Static contracts

The functionality of a contract on a connection between two ports is twofold. First of all, the contract simply monitors the data flow between the ports. Secondly, it should trigger any violation of the contract specification (for example sending too fast). This idea can easily be explained by a simple example. Look at the configuration below:

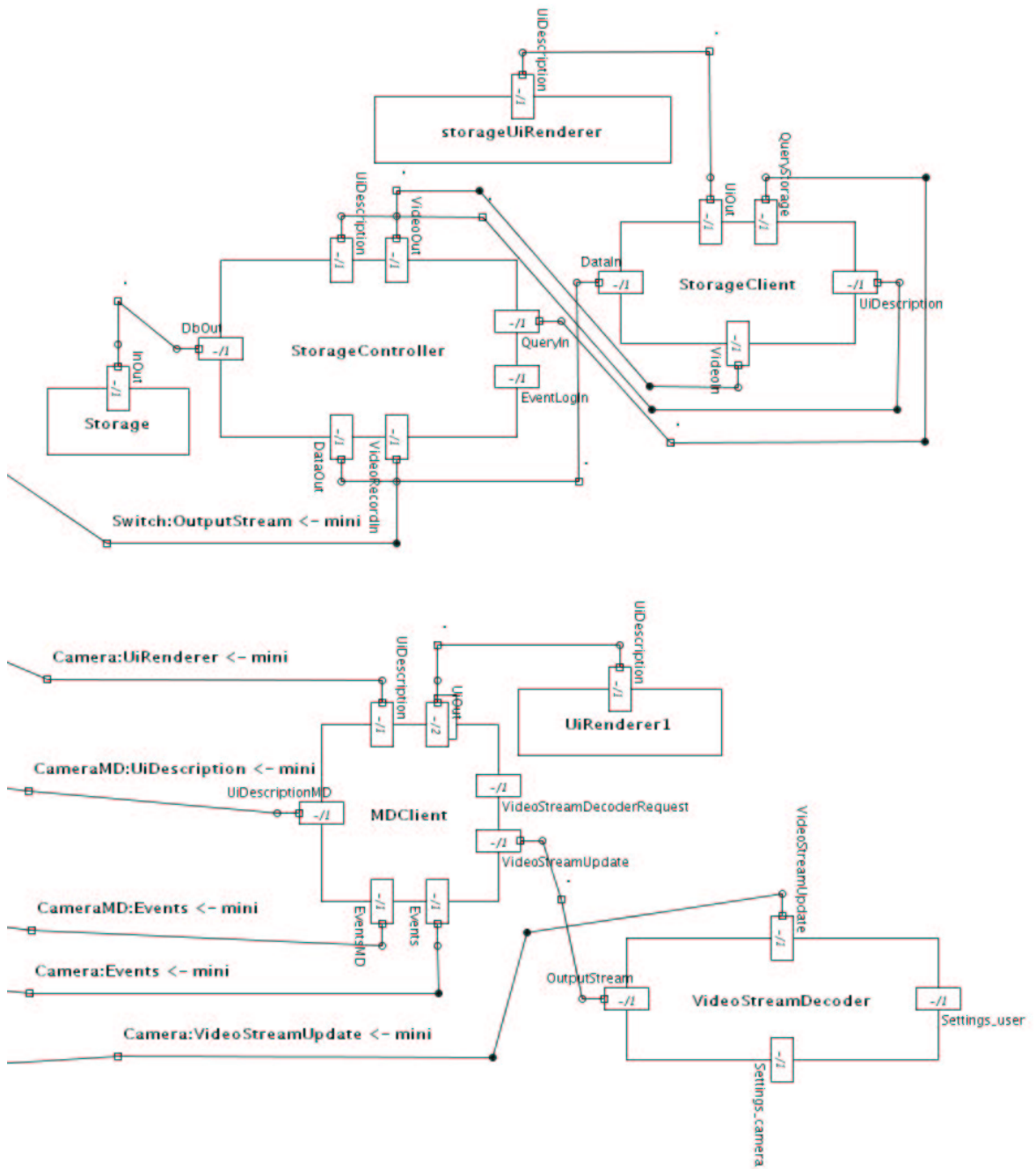


Figure 3.6: Component structure on trinity

```
COMPONENT testcases.scss.Producer hostA:producer
COMPONENT testcases.scss.Consumer hostB:consumer
```

```
CONNECT hostA:producer/data_out \
        hostB:consumer/data_in
```

It describes a simple connection between a remote producer and consumer. Bringing in a contract would make the new situation look like:

```
COMPONENT testcases.scss.Producer hostA:producer
COMPONENT testcases.scss.Consumer hostB:consumer
COMPONENT testcases.scss.ContractProducer \
        hostA:contractProducer
COMPONENT testcases.scss.ContractConsumer \
        hostB:contractConsumer
```

```
CONNECT hostA:producer/data_out \
        hostA:contractProducer/contract_in
CONNECT hostA:contractProducer/contract_out \
        hostB:contractConsumer/contract_in
CONNECT hostB:contractConsumer/contract_out \
        hostB:consumer/data_in
```

And the implementation of the contract (*ContractProducer*) could look something like:

```
componentclass ContractProducer
  implements MessageHandler
{
  port contract_in; // should be connected to the producer
  port contract_out; // should be connected to the consumer contract
  port flow_control; // for communication between the two
                      //sides of thecontracts

  ... init ...

  public synchronized void handleMessage(Port p, Message m)
  {
  if (p==contract_in) {
```

```

        ... monitoring and triggering ...
        ... communication between contracts via ...
        ... flow_control port ...
        if(allowed){
        contract_out.sendMessage(m);
        }
    }
else if (p==contract_out)
    // from consumer, just relaying message:
    contract_in.sendMessage(m);
}
}

```

The other contract looks similar, it also monitors the communication flows and gives feedback to the other contract via the *flowcontrol* port.

Contractgenerator

A new problem arises when we can not instantiate and link the contracts statically. This situation occurs whenever we load a component dynamically and link him to a remote host. That is why we implemented a contract generator component. This component is statically linked between the controller and the component that requests the dynamic creation of a component that needs contracts on one of its connections. Figure 3.7 shows this insertion of a contract generator between the controller and the component requester.

The function of the contract generator is to intercept the *createComponent* and the *createConnection* messages between the controller and the component requester. During the interception, it inserts the two extra contract components between the link that covers the remote communication.

In our camera surveillance system, such contract generators are used to generated flow control contracts between every remote connection that is used to transmit video images. Some video images are dropped whenever congestion occurs.

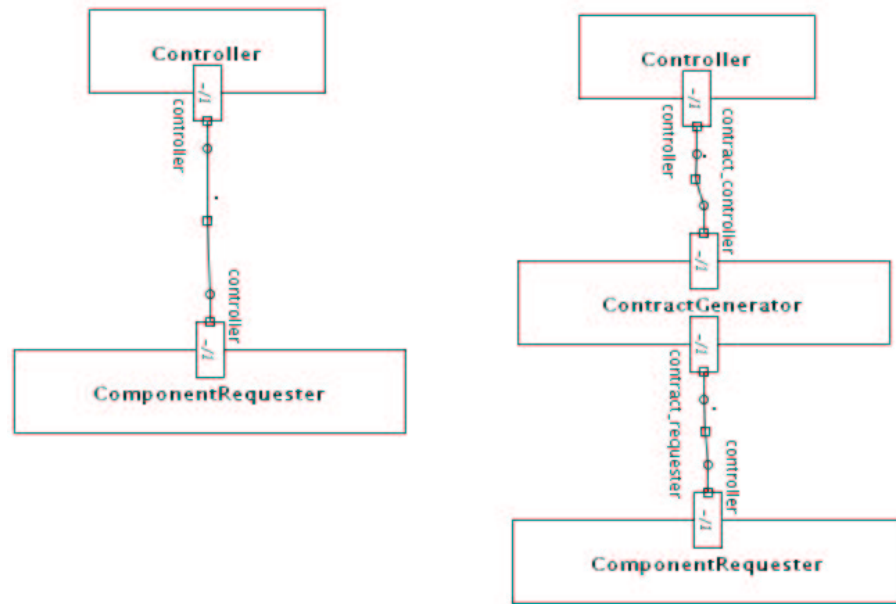


Figure 3.7: Inserting a contractgenerator

4. Conclusions

The fact that we managed to implement the design of a camera surveillance system, on the hardware we proposed, proves the component system is deployable in a real embedded environment. The implementation and the testing of the case made us be more aware of the strengths and the weaknesses of the component system. The most important weakness, undoubtedly, is the lack of quality of service mechanisms inside the system.