Reference manual and user guide

Henrik Boris-Möller, Shamiran Jaf, Jacob Karlsson, Måns Magnusson

System description

Overview

This is a surveillance system consisting of a client running on a computer and two servers running on a camera each. The client is connected to the servers through a socket connection for each server. The client is able to send commands to the servers while the servers are able to send both commands and images to the server using a data protocol.

Client

The client receives byte arrays from the sockets. Two dedicated read threads wait for images or commands to be sent through the socket from their respective server. The read threads parse the byte arrays using a data protocol which is described later. The byte arrays will either be an image or a command and will be handled differently in each case.

Object responsibilities

The relations between the objects are visualized in *figure 1*.

ReadThread - Parses byte arrays from the sockets as images or commands.
WriteThread - Writes commands to the sockets
ImageWrapper - Wraps the image byte array in an object containing the size, cameraId and timestamp of the image.
BufferMonitor - Contains the two image buffers making sure that they’re thread safe.
MainClientThread - Starts the read and write threads, establishes the socket connection and sends images from the Buffermonitor to the GUI.

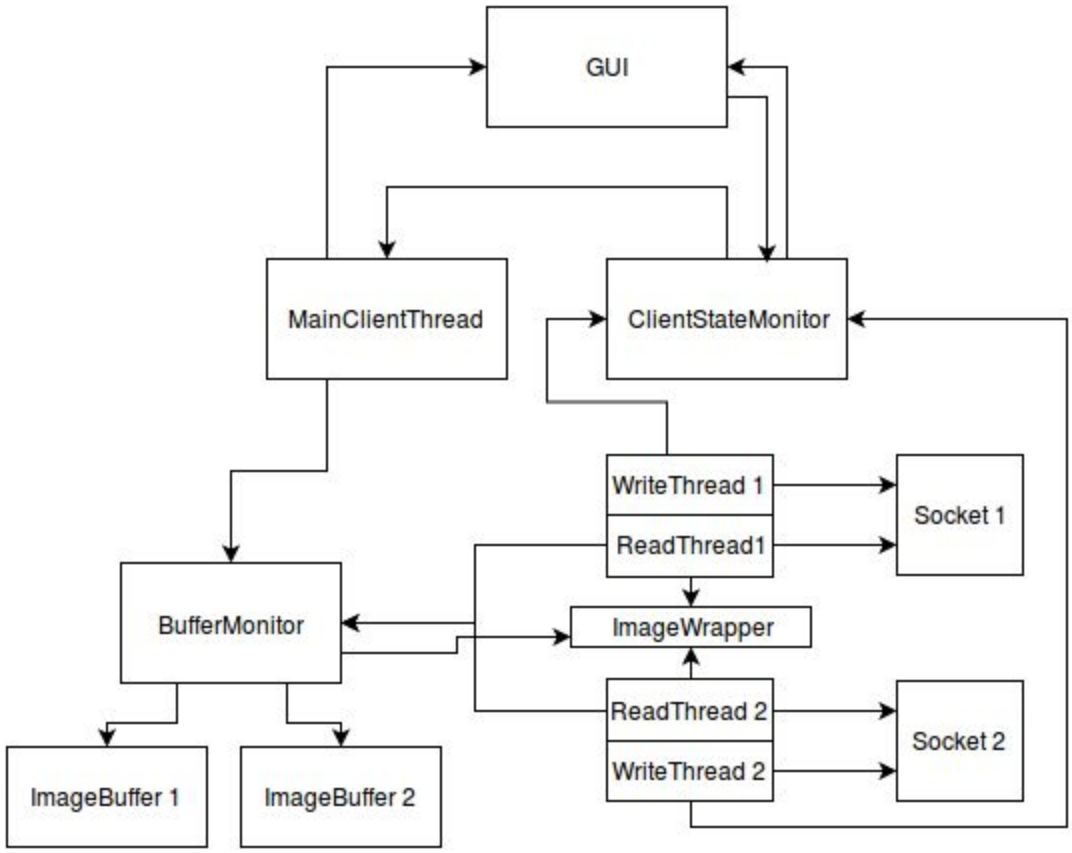


Figure 1. An arrow symbolizes a method call between the objects other than an object initialization.

Sending images from the socket to the GUI

If an image is received the read thread parses the byte array as an image and creates an ImageWrapper object which store the image data as well as information about the image size, source server id and a timestamp of when the image arrives at the server. The read thread then sends it to the BufferMonitor through a synchronized method which using the ImageWrapper server id places it in the correct ImageBuffer. The images are stored in the ImageBuffer with a ring buffer which is implemented with an array of Images (see *figure 1*).

Figure 1.

The MainClientThread will continuously request images from the BufferMonitor. If both image buffers are empty the MainClientThread will wait until it is woken up by a read thread invoking notifyAll() when depositing an image in the buffer. Otherwise the BufferMonitor will compare the oldest image of each buffer and force the MainClientThread to wait until the right time to display the image which is dependant on the synchronization state. The MainClientThread will then send the image to the GUI wrapped in a Runnable object in order to ensure that the GUI which is implemented with swing is thread-safe. *Figure 2* visualizes the flow of images from the sockets to the gui.

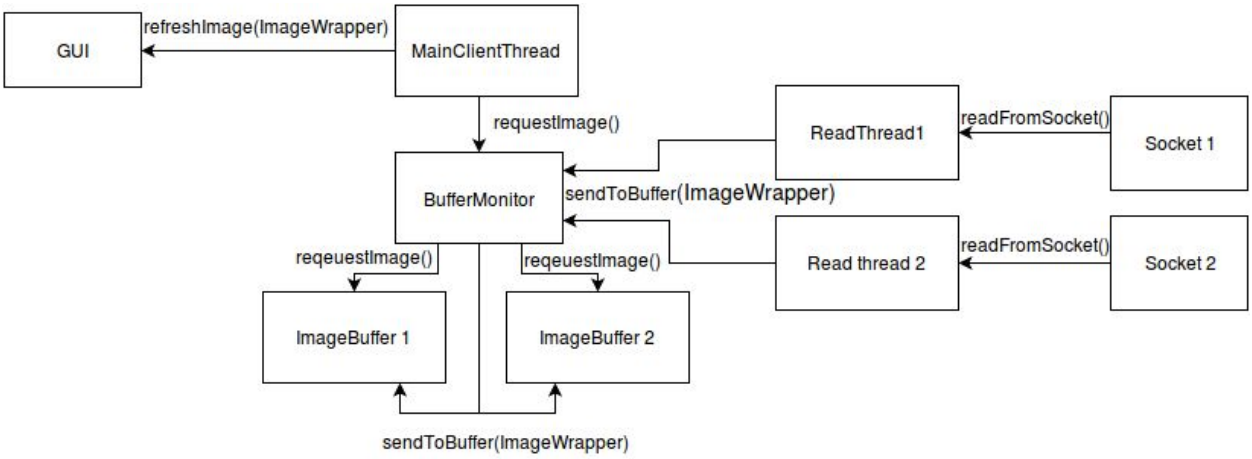


Figure 1. Model of image flow from socket to gui, irrelevant objects and methods have been omitted.

Commands

The only command from the servers that is relevant to the client is the movie mode command which is sent by the servers when motion is detected. The client will only respond to it if it is in auto mode and will in that case switch mode to movie.

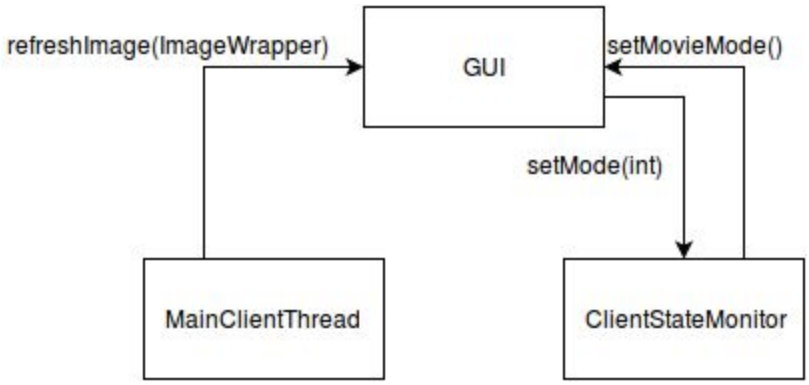
Synchronization state

The delay on each image is dependent upon the synchronization mode. There exists two modes, synchronous and asynchronous. The delay is 500 ms in the synchronous mode and 200 ms in the asynchronous mode. The system is in synchronous mode by default and transitions to asynchronous when one of the buffers is empty.

A transition from asynchronous to synchronous occurs incrementally and starts when both buffers contain at least two images. The incremental transition is achieved by incrementing the delay by 50 ms each time an image is requested by the MainClientThread until the delay is 500 ms.

GUI

The GUI is implemented in java swing and utilizes the EDT (EventDispatchThread) to display images in a thread-safe manner. The images are displayed by the EDT and sent to it in ImageWrappers by the MainClientThread through the refreshImage method. Because of the ImageWrapper containing both the time-stamp and the cameraId of the image it is able to both determine which display (JImagePanel) to display it on and calculate and display the delay of each image.



Servers

The servers send JPEG images as byte arrays to the client via socket. There are two threads, a read thread and a write thread. The write thread reads the pixel values from the camera and sends them to the socket. The only information that the server must be able to read from the socket is instructions for it to change mode. This is taken care of by the read thread which in turn updates the server's state. There are three states - idle mode, movie mode and auto mode - where each mode defines the rate at which images are sent to the server. Idle mode has a rate of 0.2 fps while movie mode has a rate of 25 fps. Auto mode is initially idle mode, but when motion is detected instructions are sent to the client (using the servers write thread) that the whole surveillance system shall change to movie mode. The client thereafter sends out instructions to all servers to change to movie mode. A UML of the server is shown below in figure 3.

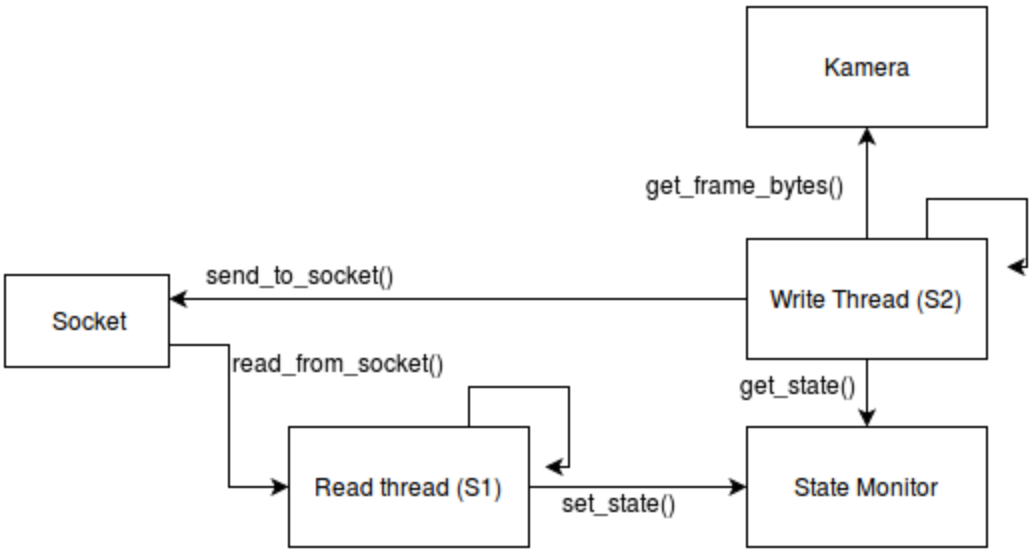


Figure 3. Shows a UML of the server side.

Data protocol

We use a data protocol which is encoded in unsigned bytes represented in a big-endian format.

<type (1 byte)> <image size (4 byte)> <timestamp (8 byte)> <image data (varying size, JPEG)>
|_obligatory__| |_____not obligatory_____|

Table 1. Shows how the different values of the type byte will be interpreted.

Type	
Value	Description
1	Data: Image
2	Command: Movie
3	Command: Idle

If type != 1 no data will be sent after the type byte.
If type = 1 the following applies:
Image size
Byte 2 to 5 will be parsed as a 32 bit integer. This integer, N, describes the image data size in bytes.

Timestamp
Byte 6 to 13 will be parsed as a 64 bit integer. This integer describes the time that the image was taken in UNIX time-format.

Image data
Byte 14 to 13+N represents the image data in JPEG format.

User guide

Setup on Ubuntu

Servers

1. Clone repository on URL https://gitlab.com/powersource/camera-server-realtime.git
2. Run “upload.bash” with input argument as an integer between 1 and 8 (input argument is the camera we are connecting to). Remember the input argument. Follow instructions on screen by confirming with “yes” and entering the password “sigge” when requested to do so. E.g. “./upload.bash 4” (“./upload.bash 5”)
3. Run “nsa-server” with the same input argument as in the previous step. E.g. “./nsa-server 4” (“./nsa-server 5”)
4. Repeat step 3 with a different input arguments. After this step the servers are ready for accepting connection from client.

Client

1. Clone repository on URL https://bitbucket.org/shamiranjaf/realtidprojektclient.git
2. Run “run.bash” with the two input arguments used when setting up the servers. E.g. “./run.bash 4 5”.

Operation

The operation of the surveillance system is uncomplicated, see figure 4. The only thing needing management is the mode control. This is in the form of a radio button group with the options “Auto”, “Idle” and “Movie”. The time delay (time between capture and display of image) for each camera is shown under its corresponding image. This is useful to see if the cameras are in synchronous mode or asynchronous mode. We have decided to define synchronous mode as a time delay of 500 ms and asynchronous mode as 200 ms. Since we have a smooth transition from asynchronous to synchronous mode this can be read of the delay field. Also under each image there is a boolean labeled “Activated Movie” which is true the client set the surveillance system into Auto mode and that camera detected movement and changed the mode to Movie. It is true for both images if the user sets the system into movie-mode. It is false for both images if the user sets the system into idle mode or if the users sets the system into auto mode and no motion is detected.

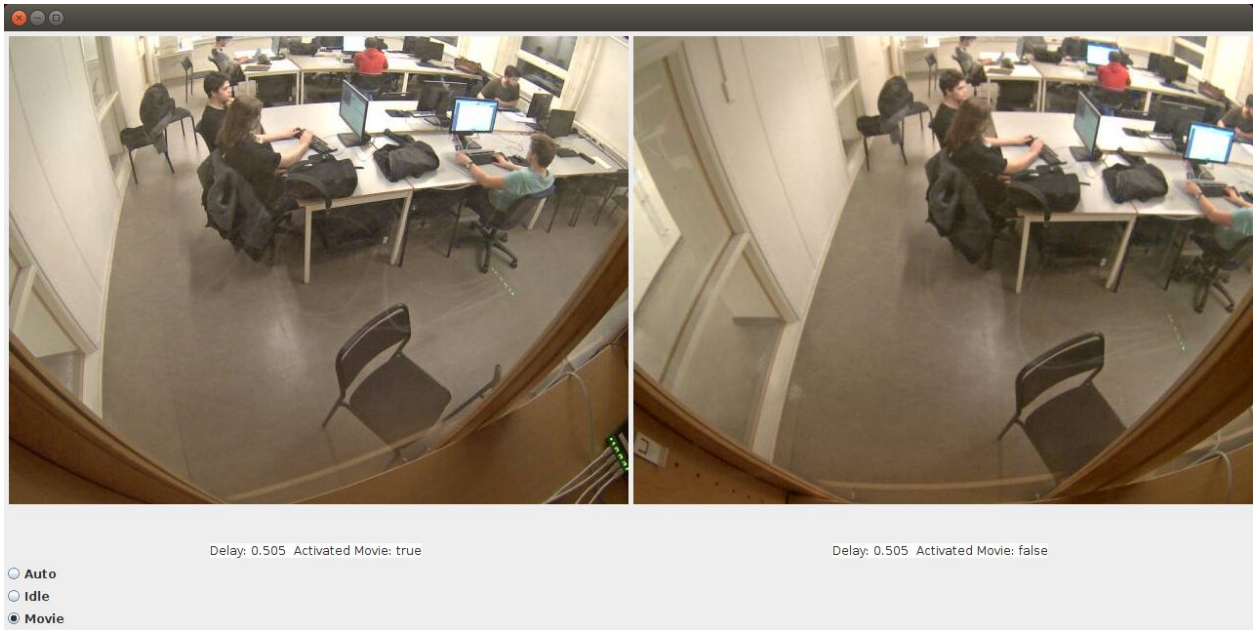


Figure 4. Shows the GUI of the client. The only control is the choice of mode.