Distributed Systems
COMP90015 2014 SM2
Project 2 - SimpleStreamer
Webcam Conference

Project 2

This project will ask you to build a more complex distributed system - a simple streaming application that can stream webcam images; called SimpleStreamer.

We will use the https://github.com/sarxos/webcam-capture library for capturing images from the webcam. An example is available to show how to read a raw image from the webcam, how to apply some crude compression (gzip) to each image, how to decompress and how to render the image in a JFrame.

There will be some emphasis on interoperability, your solution should be able to connect and interact properly with the solutions from other groups in the class. This means that the protocol and data formats need to be implemented exactly as specified in this project. If there is anything that you believe could be interpreted in more than one way, or any assumption that you need to make, then we should clarify this in class so that everyone is adhering to the same protocol and format.

Of course your implementation is up to you and will generally differ across groups, e.g. what threading model you use.

SimpleStreamer

The SimpleStreamer application is intended to allow a user to connect with an arbitrary number of other users and share their webcam image stream.

Each user should be able to make and receive connections from other users.

Multiple, simultaneous connections should be possible.

For the purpose of the assignment, there will be some limitations, specifically in the interface, as to how the connections are specified by the user. They will be specified on the command line when starting the application. A more complete application would provide a dialogue box for initiating connections, but this form of GUI is not required for this project.

Architecture

The SimpleStreamer application will be able to make connections directly to other SimpleStreamer applications and will be able to receive connections from other SimpleStreamer applications. This is effectively a peer-to-peer architecture.

Each connection will result in the webcam images being streamed both ways and being displayed on both applications.

For the purposes of the discussion, when a SimpleStreamer application initiates a connection, then we call it the client. It initiates a connection to existing instance of a SimpleStreamer application and we call this instance the server.

Protocol

We will use TCP for simplicity. We will use port 6262 as the default port, in the absence of port numbers being given on the command line.

The protocol between client and server will be quite simple.

When the client connects to the server, both the client and server will initiate a *stream*. A stream is a sequence of JSON messages written to the socket. The first message of a stream is a **StartStream** message to indicate that the stream is starting. Subsequent messages are **Image** messages, each message containing image data. When the stream is to terminate, the final message sent should be a **StopStream** message. After this message has been received, the socket can be closed.

The client and server are sending this stream simultaneously.

StartStream Message

The `startstream` message indicates that a stream of images follow.

```
{
  "type":"startstream",
  "format":"raw",
  "width":320,
  "height":240
}
```

The parameters indicate the format of the image and its size. Of course the request would be sent as a single line of text and a new line at the end.

Stream Format

In this project, the format of the streaming data is very far from ideal. It uses much more bandwidth than popular coding techniques, such as MPEG2 or H.264. However the details of efficient stream encoding are outside of the scope of this subject.

Each image will have a default resolution of 320 by 240 pixels with 3 bytes per pixel giving the RGB colours. A raw image is thus exactly 230400 bytes. If we transmitted raw images then 10 images per second would be about 2.3MB/s bandwidth. We will use the GZIP algorithm to compress each image, giving a compression ratio of about 1.7 or so. To put the compressed image into JSON requires further encoding to Base64; which unfortunately increases the size of the compressed data a little.

Each JSON message in an image stream will have this format:

```
{
  "type":"image",
  "data":"IMAGEDATAISHERE...."
}
```

StopStream Message

To terminate a stream and indicate that a socket is to be closed, the client and server should both send a **StopStream** message. Both parties should send this message for a clean close of the socket. If one party sends it (either client or server), as the result of a user terminating the program, then the other party, when receiving it, should stop sending images and send the **StopStream** message.

```
{
  "type":"stopstream"
}
```

Image streams and concurrency

The SimpleStreamer application needs to be able to service multiple connections concurrently. Its webcam image will in general be written to multiple connections.

In a simple way, at any particular time `t`, the application has an image `I(t)` in memory, read from the webcam. As time progresses, the image in memory changes (as it is being replaced by a new image from the webcam). Whenever it is time to give an image to a client (e.g. at time `t'`) then image `I(t')` is given; whatever that happens to be.

Displaying images

The **StartStream** message includes the format and image size for the image. When displaying the image, the application must correctly interpret it. In this project the format will always be `"raw"` and it will always be encoded in the JSON message in the same way (as explained earlier). However the image size may vary, as the user will have the opportunity to set this on the command line. Smaller images will use less bandwidth. Therefore the application needs to use the image size from the **StartStream** message when displaying the image.

Rate Limiting

Generally, the rate at which images can be written to the socket will be determined by the available bandwidth at the sender and receiver and intermediate network.

Using TCP, the write operation will block when the local buffer (in the OS) becomes full. This will become full as the OS waits for existing data to be reliably sent to the other side. This will naturally limit the sending *frame rate*. The sending thread will be blocked while it waits to write the remaining part of the image message. Serving concurrent sockets should reduce the frame rate for each socket uniformly; your implementation should ensure that each socket is fairly serviced.

In this project, the user will have the ability to start the application with an image size. A small image size will generally allow higher frame rates.

However naively writing images at maximum rate on all sockets will still saturate the available bandwidth. Therefore each user will also be able to provide a rate limit in the form of *sleep interval*. Images written to a socket should have a pause between them of duration equal to the *sleep interval*, i.e. between completing an image message write to the TCP socket, to when the next image message write is started.

Using a combination of image size and rate limiting, a user can attempt to obtain an acceptable frame rate. The default rate limit should be 100ms.

Command line

Let's all use Java 1.7.

The command line execution is:

```
java -jar SimpleStreamer.jar [-sport X] [-remote hostname1,hostname2,...
    [-rport Y1,Y2,...]] [-width W] [-height H] [-rate Z]
```

Where `sport` is the server port to use, defaulting to 6262 if no `sport` is given, `remote` specifies a comma separate list of `hostname` to connect to, `rport` specifies a comma separated list of port numbers for the remote servers, `width` and `height` are the desired image parameters and `rate` specifies a sleep time that is desired. For example:

```
java -jar SimpleStreamer.jar
```

will not connect to anyone and will use the default 6262, waiting for connections.

```
java -jar SimpleStreamer.jar -remote sunrise.cis.unimelb.edu.au,
    sundowner.cis.unimelb.edu.au -rport 2323,6262
```

will try connecting to two other SimpleStreamer applications, with default parameters otherwise.

Report

- Use 10pt font, double column, 1 inch margin all around.
- Write 2 to 3 pages addressing at least the following topics in detail:
    - Bandwidth consumption, what ways could be used to reduce bandwidth? Give details.
    - Security, what security issues exist and how could the protocol be changed to address them?
    - Scalability and rate limiting, why is rate limiting done? does it achieve scalability?
    - Any other improvements or observations that you would like to discuss.

Submission

You need to submit the following via LMS:

- Your report in PDF format *only*.
- Your SimpleStreamer.jar application.
- Your source files in a .ZIP or .TAR archive *only*.

Submissions will be open and due in Week 12.