Chapter 36. Networked Cameras

In this chapter I'll develop a TCP/IP client/server cameras application: the clients send images obtained from their webcams to the server to be centrally displayed. The idea is illustrated in Figure 1.

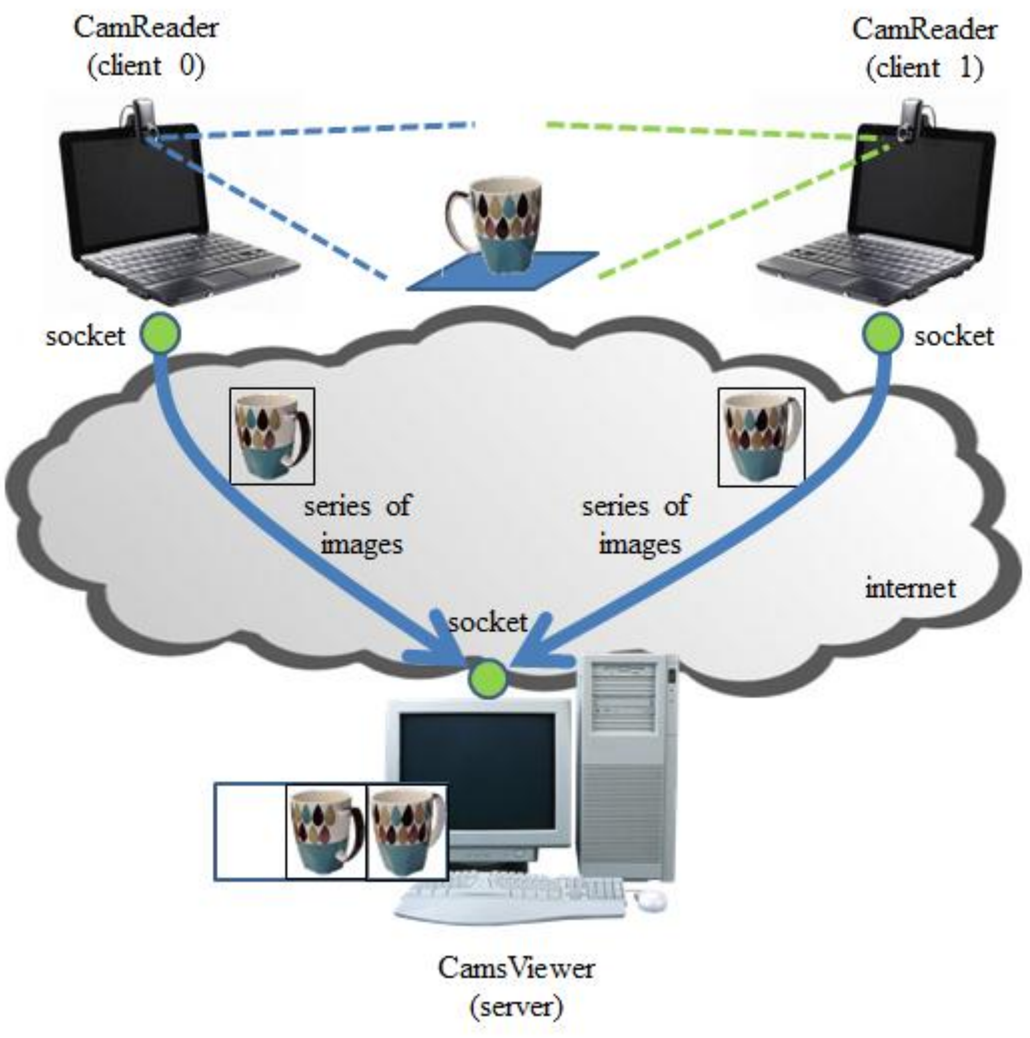

Figure 1. Networked Cameras.

A novel feature of the application is that a client divides its current camera image into subimages before passing them to the server. The client compares the subimages with ones obtained from the previously snapped image, and only those parts which have changed are sent. This approach is shown in Figure 2, with the camera's image divided into 16 parts.

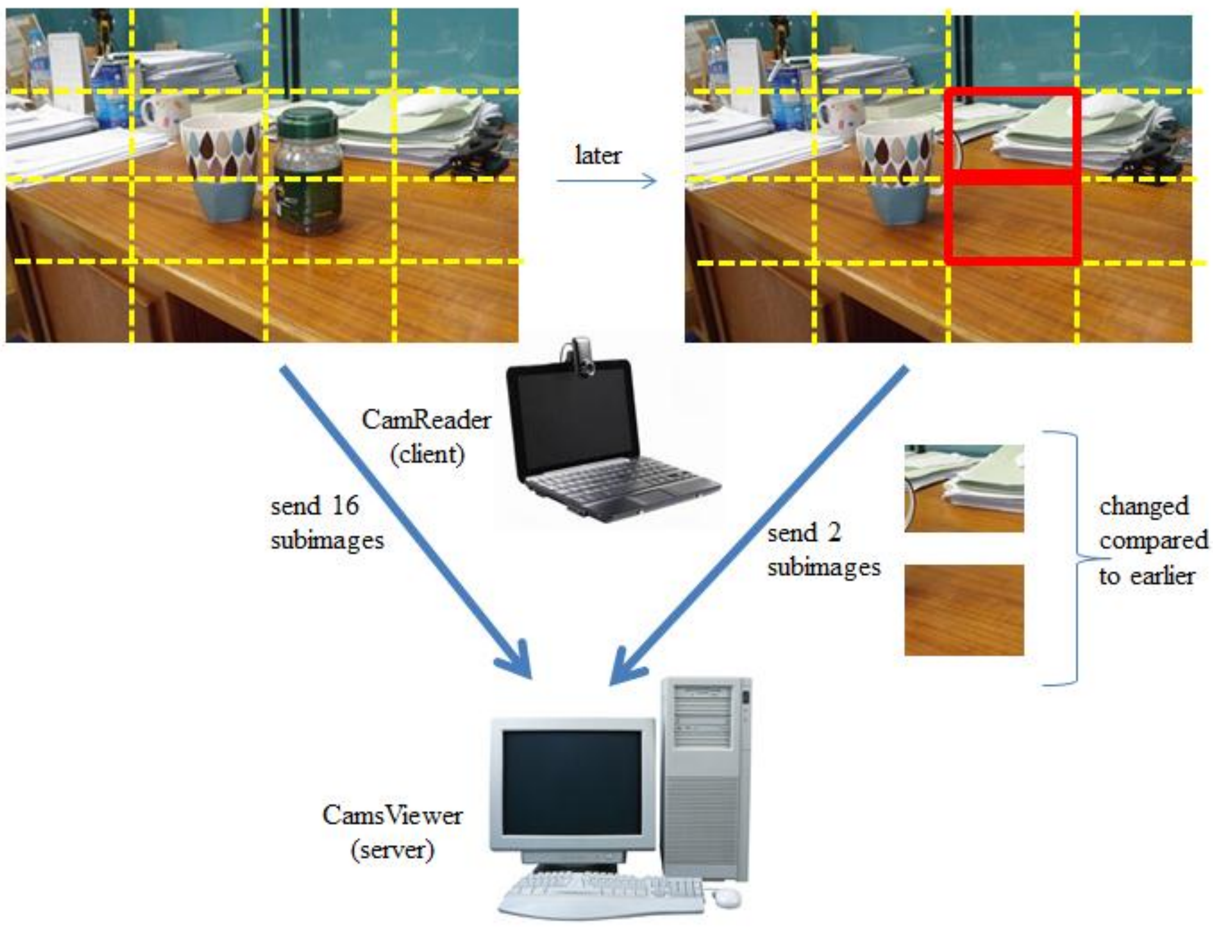

Figure 2. Dividing and Sending Subimages.

On the left of Figure 2, all the subimages are sent to the server by the client, and the it stitches them together before displaying the complete picture. However, the next client image (on the right of Figure 2) only contains two changed areas, so only those need to be sent this time. The advantage is a reduction in the amount of network communication, related to how much the client's view of a scene changes from frame to frame. The downside is that the server will need to retain previous subimages, so they can be combined with the new ones.

Figures 1 and 2 are somewhat misleading since the clients don't really send subimages to the server. The client transforms each image into a sequence of bytes, and sends those, together with an ID number identifying the subimage.

Figure 3 is a screenshot of the server, called CamsViewer, displaying two views of the author. The third panel is white since there's no third client connected to the server at the moment.

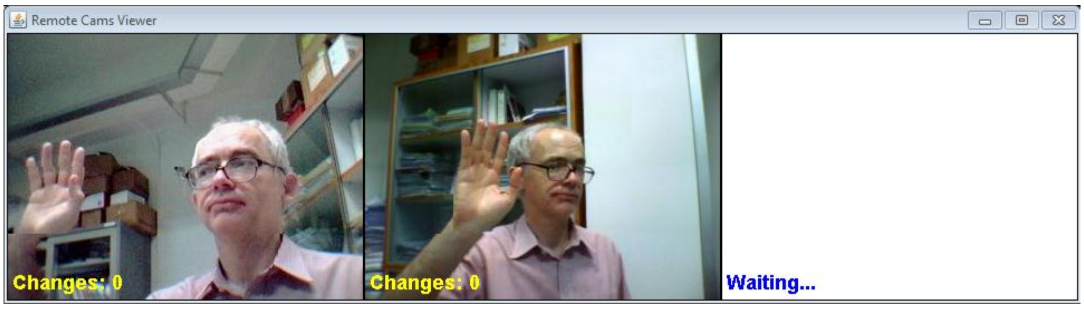

Figure 3. The CamsViewer Server Displaying Two Clients.

Along with the clients' images, the server displays the number of changed subimages used to build each image. The user hasn't moved in Figure 3 since the previous camera snaps (around 100 milliseconds before), so both client images show zero changes.

One problem with the subimage approach is that a reconstituted image may contain slightly out-of-date parts, as in Figure 4.

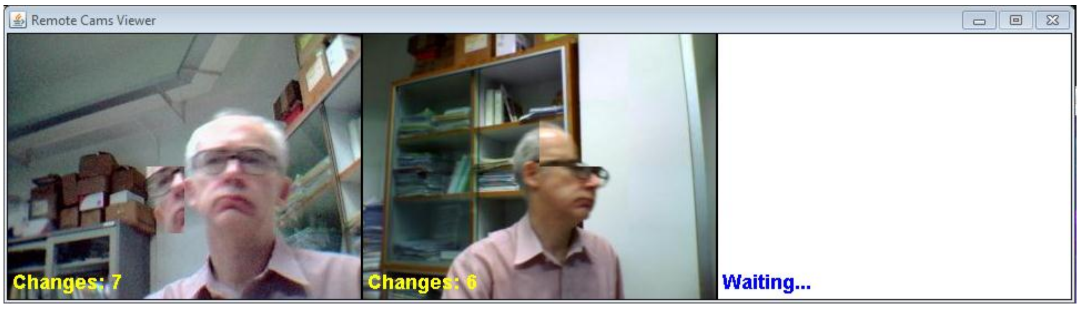

Figure 4. The CamsViewer Showing Some old Subimages.

The two images have been updated with 7 and 6 subimages respectively, but one subimage in each client is still out-of-date. In practice, these old parts are usually replaced within a second or so after the movement has occurred. Part of the reason for this problem is that I'm using a simple brightness measure to compare old and new subimages, as explained below.

1. Client/Server Structure

The class diagrams for the client-side of the application are shown in Figure 5.

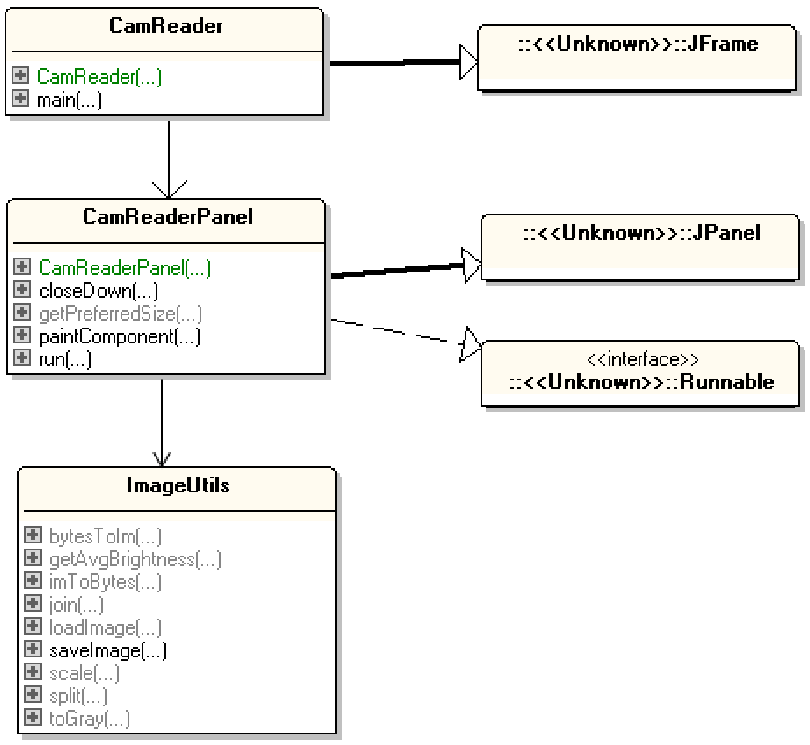

Figure 5. CamReader: the Client-Side of the Application.

Each client machine runs its own copy of CamReader, which creates a display panel in a window to show the current camera input. This input is read in a frame at a time by the JavaCV FrameFrabber class. A CamReader window is shown in Figure 6.

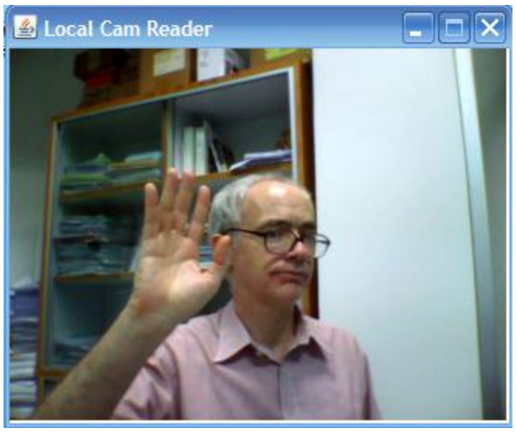

Figure 6. A CamReader Instance.

The CamReaderPanel acts both as a rendering area for the grabbed images and as a TCP/IP client with a socket link to the server. Various image processing functions, including ones for splitting images into parts, converting subimages to bytes and back again, and gluing subimages together, are stored as static methods in the ImageUtils class.

The class diagrams for the server side of the application are shown in Figure 7.

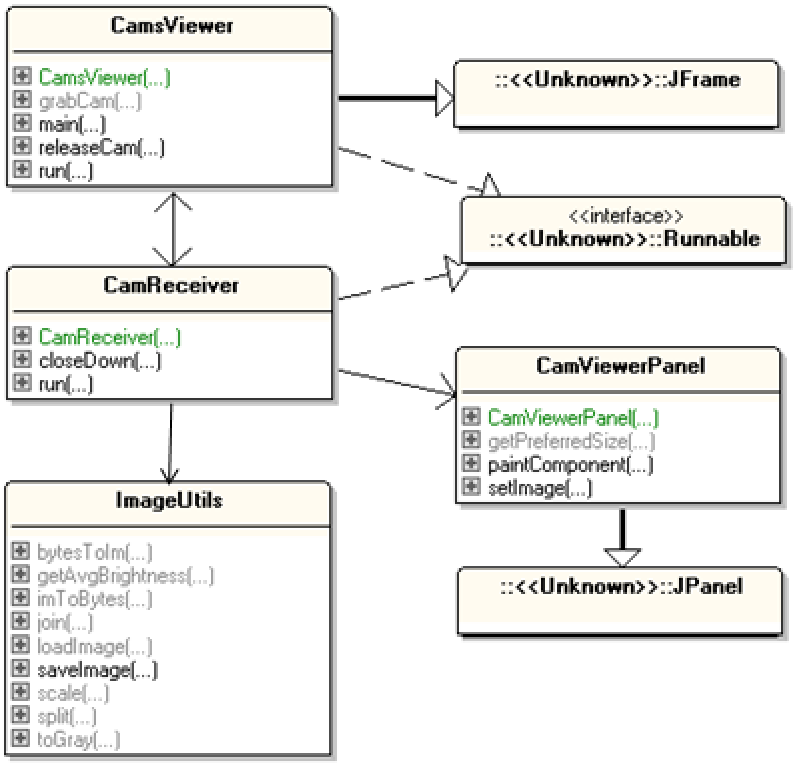

Figure 7. CamsViewer: the Server-Side of the Application.

The top-level class, CamsViewer, creates a window with a ServerSocket to receive connections from clients. A new client socket triggers the creation of a CamReceiver instance which is threaded so it can wait on data arriving from the client without freezing the CamsViewer server. Each CamReceiver is assigned a CamViewerPanel instance, where the reconstructed images coming from the client are displayed. CamsViewer limits itself to creating three CamViewerPanels, which means that it can only handle three clients concurrently. Image processing features are again supported by static methods in ImageUtils, the same class as used by the clients.

Figure 8 is a redrawing of the networked camera diagram of Figure 1 to show where various classes are utilized.

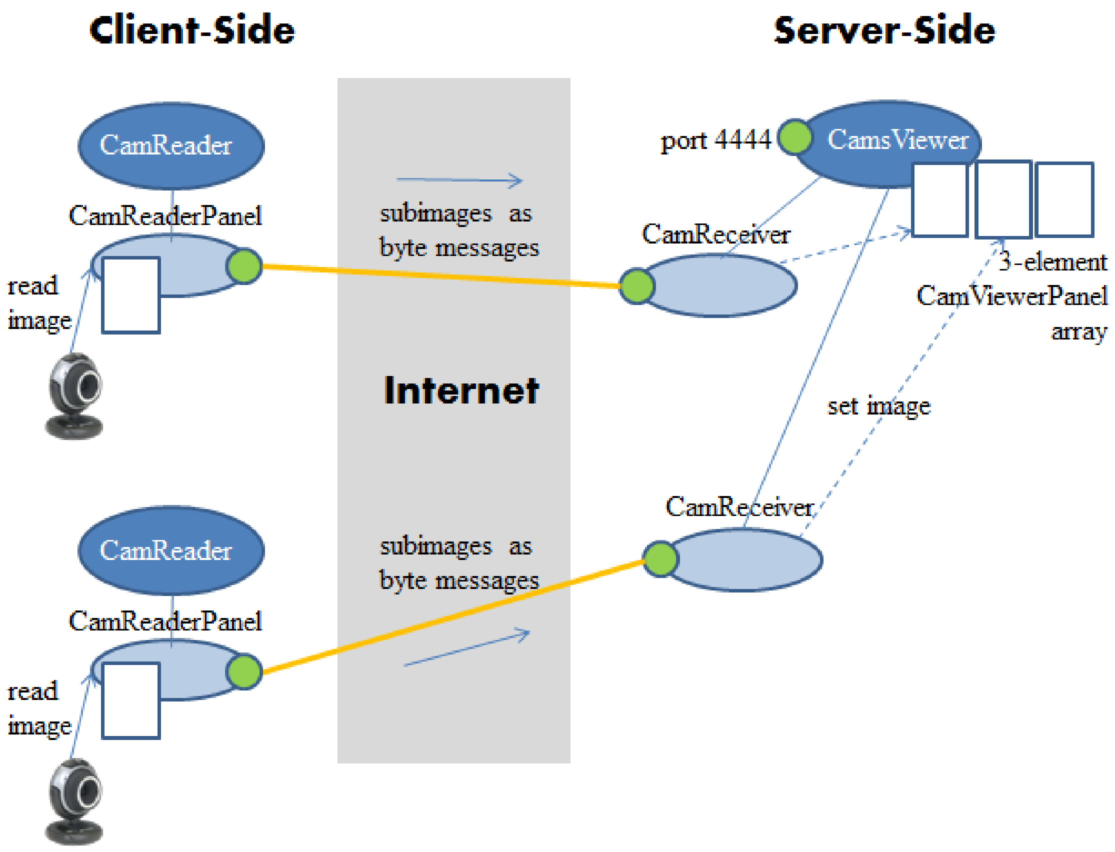

Figure 8. The Networked Cameras and their Classes.

2. The Camera Client

The heart of the CamReaderPanel class is a run() method which starts by creating a FrameGrabber instance and a socket link to the server. Then it enters a loop which snaps a picture, divides it into subimages, decides which parts have changed, and sends those to the server.

```
// globals
private static final int DELAY = 100;  // ms, time between snaps
private static final int CAMERA_ID = 0;

private static final int PORT = 4444;   // server port
private static final int NUM_SEPS = 4;
          // subimage separations per row and column

private static final int END_TRANS = -99;

private String ipAddr;    // IP address of CamsViewer
private BufferedImage image = null;   // scaled image
private volatile boolean isRunning;
private volatile boolean isFinished;


public void run()
{
  FrameGrabber grabber = initGrabber(CAMERA_ID);
  if (grabber == null)
    return;

  IplImage snapIm = picGrab(grabber, CAMERA_ID);
  if (snapIm == null)
    return;
  double scaleFactor = calcScale(snapIm.width(), snapIm.height());

  long duration;
  isRunning = true;
  isFinished = false;
  BufferedImage[] imParts;    // holds parts of the total image

  try {
    // connect to CamsViewer via a socket
    System.out.println("Connecting to " + ipAddr +
                                 ", port: " + PORT + "...");
    Socket sock = new Socket(ipAddr, PORT);
    DataOutputStream dos =
                new DataOutputStream(sock.getOutputStream());

    while (isRunning) {
      long startTime = System.currentTimeMillis();

      snapIm = picGrab(grabber, CAMERA_ID);
      image = ImageUtils.scale(snapIm.getBufferedImage(),
                                 scaleFactor);    // scale input
      imParts = ImageUtils.split(image, NUM_SEPS);  // split
      updateParts(dos, imParts);
                     // send the parts that have changed
      repaint();

      duration = System.currentTimeMillis() - startTime;
      if (duration < DELAY) {
        try {
          Thread.sleep(DELAY-duration);
                 // wait until DELAY time has passed
        }
        catch (Exception ex) {}
      }
    }

    camera.close();    // close down the camera
    sendMessage(dos, END_TRANS, null);
            // tell CamsViewer that this client is finishing
    sock.close();
    System.out.println("Reader shut down");
    isFinished = true;
  }
  catch(IOException e)
  {  System.out.println("Could not connect to server");
     System.exit(1);
  }
}  // end of run()
```

The CamReader client is supplied with the IP address of the server as a command line argument, but the server's port (4444) is a hardwired constant.

Aside from the use of subimages to reduce communication overheads, run() illustrates two other load-reduction techniques. The while-loop is set to iterate every DELAY

(100) milliseconds, which means that around 10 pictures are snapped each second, processed, and sent to the server. The quantity of data sent out can easily be reduced if the DELAY value is increased. The other data reduction technique is image scaling, which is hardwired to reduce each picture to 320 by 240 pixels.

The ImageUtils.split() method returns an array of subimages by dividing the original image into NUM_SEPS rows and columns using BufferedImage.getSubimage(). The resulting array contains NUM_SEPS*NUM_SEPS images which are analyzed and posted out by updateParts().

```
// globals
private static final int BRIGHT_THRESHOLD = 150;
       // used to judge if two brightnesses are different

private static final int END_UPDATE = -98;

private int[] brightness;   // brightness data for each subimage part


private void updateParts(DataOutputStream dos,
                                      BufferedImage[] imParts)
{ byte[] imPartBytes;
  for(int i=0; i < imParts.length; i++) {
    int currBright = ImageUtils.getAvgBrightness(imParts[i]);
    if (Math.abs(currBright - brightness[i]) >= BRIGHT_THRESHOLD) {
      // brightnesses are different
      imPartBytes = ImageUtils.imToBytes(i, imParts[i]);
      sendMessage(dos, i, imPartBytes);
                   // send message containing the image bytes
      brightness[i] = currBright;      // store brightness level
    }
  }
  sendMessage(dos, END_UPDATE, null);
             // all updates for this image have been sent
}  // end of updateParts()
```

The code calculates the average brightness for a subimage, and compares it with the brightness of the corresponding subimage from the previous iteration. If the two brightnesses differ by BRIGHT_THRESHOLD or more then the new subimage is sent to the server as a sequence of bytes.

I 'borrowed' the brightness calculation method from a blog post at http://mindmeat.blogspot.com/2008/07/java-image-comparison.html – it converts the subimage to a grayscale, sums the grayscale values in all the pixels, and averages the total.

The ImageUtils.toGray() method used by ImageUtils.getAvgBrightness() could be applied to the entire image as another way of reducing its size. However, I decided to stay with transmitting colored images.

My sendMessage() method implements a simple communication protocol between the client and server based on messages of the form:

```
      <part index> <byte array length>  <image data bytes ...>
```

and the sending of the special messages END_UPDATE and END_TRANS. END_UPDATE means that all the updates for the current image have been sent, while END_TRANS signals that the client is closing down.

In the image data message, the <part index> argument identifies which subimage is being sent. The byte array length is employed on the server-side to convert the correct number of bytes back to an image. ImageUtils contains imToBytes() and bytesToIm() methods for converting a subimage to a bytes sequence, and back again.

The sendMessage() method:

```
// globals
// special message IDs
private static final int END_UPDATE = -98;
private static final int END_TRANS = -99;

private static final int MAX_WRITE_FAILS = 50;

private int numWriteFails = 0;


private void sendMessage(DataOutputStream dos, int idx, byte[] bytes)
{
  try {
    if (idx == END_UPDATE)
      dos.writeInt(idx);
    else if (idx == END_TRANS) {
      System.out.println("Sending client END message");
      dos.writeInt(idx);
    }
    else {   // send image part
      dos.writeInt(idx);
      dos.writeInt(bytes.length);
      dos.write(bytes, 0, bytes.length);
    }
  }
  catch(IOException e)
  {
    numWriteFails++;
    if (numWriteFails >  MAX_WRITE_FAILS) {
      System.out.println("Terminating due to multiple
                                    server write fails");
      System.exit(1);
    }
  }
}  // end of sendMessage()
```

sendMessage() uses a global numWriteFails counter to record the number of failed writes to the server socket. If the total exceeds MAX_WRITE_FAILS, then the client issues an error message and terminates.

3. The Cameras Server

CamsViewer carries out two main tasks: it's a threaded server that spawns a CamReceiver thread to deal with each new client, and it’s a JFrame that contains multiple CamViewerPanel panels. Each new CamReceiver instance is paired with a CamViewerPanel but, to keep things simple, there's a fixed number of

CamViewerPanel objects. If all of these panels are being used (by existing receivers) then a new client connection is rejected by CamsViewer.

When a receiver terminates, it hands its panel back to the server so it can be used for another client when one connects.

The CamsViewer constructor:

```
// globals
private static final int NUM_PANELS = 3;

private CamReceiver[] receivers;
            // receives image parts from a CamReader client

private CamViewerPanel[] camsPan;
            /* each one displays the reconstituted image
               for a matching CamReceiver */

private boolean[] usedCams;
           // which camera panels are currently in use?


public CamsViewer()
{
  super("Remote Cams Viewer");
  Container c = getContentPane();
  c.setLayout( new BoxLayout(c, BoxLayout.X_AXIS));
        // the panels are laid out across the screen

  camsPan = new CamViewerPanel[NUM_PANELS];
  usedCams = new boolean[NUM_PANELS];
  receivers = new CamReceiver[NUM_PANELS];

  for (int i=0; i < NUM_PANELS; i++) {
    camsPan[i] = new CamViewerPanel();
        // the reconstructed images appear in these panels
    usedCams[i] = false;
       // all the panels are available (i.e. not in use) at the start
    c.add( camsPan[i]);
  }

  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { for(CamReceiver cr : receivers)
        if (cr != null)
          cr.closeDown();
               // stop receiving images from CamReader clients
      System.exit(0);
    }
  });

  setResizable(false);
  pack();

  // position this window at the bottom middle of the screen
  Dimension scrSize = Toolkit.getDefaultToolkit().getScreenSize();
  int x = (scrSize.width  - getSize().width)/2;
  int y = scrSize.height - getSize().height - 30;
  setLocation(x, y);
  setVisible(true);
  new Thread(this).start();   // start waiting for client connections
} // end of CamsViewer()
```

The receivers and their panels are paired up by being stored in corresponding positions in the receivers[] and camsPan[] arrays. Since the camera panels can be reused, it's also useful to have a usedCams[] array to maintain a list of which panels are in use, and which are free.

One reason for keeping the number of panels small is that it simplifies the layout task for the window, which uses a horizontal box layout (see Figure 3).

3.1. Creating a Receiver

CamsViewer's run() method consists of a loop which wait for a client to contact it. The resulting socket is passed to a newly instantiated CamReceiver object only if there's a camera panel available for displaying the received images.

```
// globals
private static final int PORT = 4444;


public void run()
{
  Socket sock;
  try {
    ServerSocket servsock = new ServerSocket(PORT);
    printHostInfo();
    while (true) {
      System.out.println("Waiting for a cam connection...");
      sock = servsock.accept();
      /* if there's a free camera panel then pass
         it to a new receiver for the client */
      int camIdx = grabCam();
      if (camIdx != -1) {   // a camera panel is available
        receivers[camIdx] =
            new CamReceiver(sock, camIdx, camsPan[camIdx], this);
        System.out.println("--assigned new cam to panel " + camIdx);
      }
      else {
        System.out.println("Not enough panels to accept a new cam");
        sock.close();
      }
    }
  }
  catch (IOException e)
  {  System.out.println("CamsViewer network problem");
     System.exit(0);
  }
} // of run()
```

The selection of an available camera panel is handled by getCam(), which is defined as a synchronized method.

```
// globals
private boolean[] usedCams;
```

```
public synchronized int grabCam()
// return the index of a free camera panel, or -1
{
  for (int i=0; i < NUM_PANELS; i++)
    if (!usedCams[i]) {
      usedCams[i] = true;
      return i;
    }
  return -1;
}  // end of grabCam()
```

The reason for the synchronization is that the usedCam[] array can also be manipulated by a terminating CamReceiver instance, which is running in its own thread. It calls releaseCam() to set a specified usedCams[] entry to false to indicate that the panel with the same index is available again.

```
public synchronized void releaseCam(int idx)
{
   if ((idx >= 0) && (idx < NUM_PANELS)) {
    usedCams[idx] = false;
    receivers[idx] = null;
  }
}  // end of releaseCam()
```

4. Processing a Client's Data

A CamReceiver waits for a CamReader client to send image parts to it via a TCP/IP socket. The subimages are combined into a single image and displayed in an CamViewerPanel panel. Hardwired into CamReceiver is the knowledge that NUM_SEPS*NUM_SEPS subimages form a single image, and how those subimage are arranged to form a whole.

CamReceiver utilizes a threaded loop to wait for subimage update messages. During an image's update less than NUM_SEPS*NUM_SEPS subimages may arrive, in which case the image's reconstruction must use subimages delivered during earlier updates. An update is finished when CamReceiver receives an END_UPDATE message. It then passes the constructed image to CamViewerPanel to be displayed, and goes back to waiting for the next collection of updates.

If CamReceiver receives an END_TRANS message then its client is terminating, and it should also terminate after making its CamPanel available to the top-level server, for another receiver to use.

The looping behavior is located in CamReceiver's run() method:

```
// globals private static final int NUM_SEPS = 4;
              // image separations per row and column

private Socket sock;
private int recID;                // ID for this receiver
private CamViewerPanel camPanel;
                     // the display panel for this receiver
```

```
private CamsViewer camsViewer;    // the top-level server

private int numPartsUpdated = 0;

private volatile boolean isRunning;


public void run()
{
  BufferedImage[] imParts = new BufferedImage[NUM_SEPS*NUM_SEPS];
   try {
     DataInputStream dis =
              new DataInputStream(sock.getInputStream());
    isRunning = true;
    while (isRunning) {
      if (!updateParts(dis, imParts))  // update subimages
        break;

      BufferedImage im = ImageUtils.join(imParts, NUM_SEPS);
                              // join subimage parts together
      if (im == null)
        System.out.println("Image is null");
      else
        camPanel.setImage(im, numPartsUpdated);
    }

    // Close this connection (not the overall server socket)
    System.out.println("Cam receiver " + recID + " terminating");
    camPanel.setImage(null, 0);
    camsViewer.releaseCam(recID);
            /* release the CamViewerPanel so it can
               be used by another receiver */
    sock.close();
  }
  catch(IOException ioe)
  {  System.out.println(ioe); }
} // end of run()
```

The while-loop inside run() can terminate in a number of ways. It may be stopped by isRunning being set to false, which is done via a call to closeDown() from the top-level CamsViewer server. Other possibilities are that updateParts() detects a socket error of some kind, or receives an END_TRANS message, both of which cause it to return false.

The main job of updateParts() is to read in new subimages and update the corresponding entries in the imParts[] array:

```
// globals
// special message IDs
private static final int END_UPDATE = -98;
private static final int END_TRANS = -99;


private int numPartsUpdated = 0;


private boolean updateParts(DataInputStream dis,
                                     BufferedImage[] imParts)
{ numPartsUpdated = 0;
```

```
  while (numPartsUpdated < imParts.length) {
    int res = readMessage(dis, imParts);
    if (res == END_UPDATE)
      break;
    else if ((res == END_TRANS) || (res == -1))
      return false;
    numPartsUpdated++;
  }
  return true;
}  // end of updateParts()
```

updateParts() repeatedly calls readMessage() to read in a message and convert its data into an image. This continues until all of imParts[] has been updated or an END_UPDATE message is received. It's also possible for the update to fail, which is signaled by readMessage() returning END_TRANS or -1.

readMessage() has to deal with three types of message:

`<part index> <byte array length>  <image data bytes ...>`,
END_UPDATE, and END_TRANS.

readMessage() either returns the index of the required subimage, END_UPDATE, END_TRANS, or -1 for an error (such as a socket read error):

```
private int readMessage(DataInputStream dis,
                                 BufferedImage[] imParts)
{ try {
    int idx = dis.readInt();
    if (idx == END_UPDATE)
      return END_UPDATE;   // no more parts to update on this round
    else if (idx == END_TRANS) {
      System.out.println("Remote client has disconnected");
      return END_TRANS;     // time to finish reading
    }

    int len = dis.readInt();
    if (len == 0) {    // no data in this part
      System.out.println("Part " + idx + " has no data");
      return -1;   // no data read
    }
    else {    // read in the image data
      byte[] imBytes = new byte[len];
      dis.readFully(imBytes);
      imParts[idx] = ImageUtils.bytesToIm(idx, imBytes);
      return idx;   // data read for idx subimage
    }
  }
  catch (InterruptedIOException iioe)
  { System.out.println("Remote client connection timed out");
    return -1;   // time to finish
  }
  catch(IOException e)
  { System.out.println("Client socket read error");
    return -1;   // time to finish
  }
}  // end of readMessage()
```