Development of a Remote Object Webcam Controller (ROWC) with CORBA and JMF

A Thesis submitted
to the Graduate School
University of Arkansas at Little Rock

in partial fulfillment of requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

in the Department of Computer Science
of the Donaghey College of Information Science and Systems Engineering

July 2002

Frank McCown

BS, Harding University, Searcy, Arkansas, 1996

Abstract

The need for a low-cost, scalable, cross-platform, distributed system for performing visual surveillance prompted the development of the Remote Object Webcam Controller (ROWC). With a web cam and personal computer, a user can use ROWC to remotely monitor live video and archive video for later retrieval. This thesis describes the use of the Java™ Programming Language, Java Media Framework™ (JMF), both developed by Sun Microsystems, and the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) for constructing the ROWC system. A comparison of this system with other similar distributed systems is made, and several enhancements to the system are offered for future research. Our experience in building ROWC has uncovered some JMF bugs, but the combination of CORBA and JMF has been shown sufficient for producing a distributed video surveillance system that is portable, flexible, and extendable.

Keywords: Java, JMF, CORBA, remote object, distributed system

Table of Contents

List of Figures

Acknowledgements

I would like to thank Dr. Yeong-Tae Song for providing me the opportunity to learn about distributed programming during my summer researching position. I'd also like to thank Dr. Charles Wesley Ford, Jr., my thesis advisor, for his helpful guidance and feedback during the writing of this thesis. Thank you to the professors who served on my thesis committee, for the time they have invested in ensuring my thesis is thorough and defendable.

Great appreciation goes to my colleagues at Harding University who have supported me during the last several years while I taught full time and worked on my Masters degree. Special thanks goes to Dr. Tim Baird, my department chairman, to whom this thesis is dedicated. Without Tim's personal commitment and backing I would not have had the opportunity to teach at Harding University. It has been a wonderful experience to be at Harding as both a student and a faculty member, and I have learned from my fellow colleagues what it means to be a Christ-led college instructor.

Thank you to my close friends who have been a source of strength and who always provide a much-needed escape from the world of academia. Thanks to my family for their continual love and support. And all praise goes to God through whom all blessings flow. As I am continually reminded, "the fear of the Lord is the beginning of wisdom." (Psalms 111:10)

Dedication

This thesis is dedicated to Dr. Tim Baird, my boss and mentor, for his continual encouragement to me while teaching and completing my masters degree. The opportunity to teach at Harding University has been a great blessing.

Introduction

There is a growing need today to visually monitor an area from a remote location using a video camera or web cam. Distributed multimedia applications of this type require an efficient method of transporting real-time video from monitors to listening clients. Streaming multimedia frameworks have become increasingly popular since the Internet is used as a media transportation layer. Advances in CPU processing power and increased network bandwidth have contributed to this growth [MSS02]. Working with streaming video presents several challenges, including interfacing with a variety of web cams on different operating systems [Sun02-4], efficiently streaming captured video [WHZ01], archiving video streams [PHS96], and timely access to such archives [Gem95].

This thesis describes the use of the Java™ Programming Language, Java Media Framework™ (JMF), both developed by Sun Microsystems, and the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) for constructing a low-cost, scalable, cross-platform, distributed system to perform surveillance. A comparison of this system with other similar distributed systems will also be made. Several enhancements to the system will be offered for future research.

Motivation

The proliferation of web cams and the increasing use of streaming media have improved the ability of users to communicate with each other, to be entertained and to monitor or perform surveillance. Naturally, the development of software for providing such services has become prevalent in the software development community. The popular Java Programming Language has continued to establish itself as an ideal language for building cross-platform, object-oriented software. JMF allows Java programs a vendor-independent and platform-independent interface to a variety of web cams. It also provides mechanisms for transporting streaming video using the Real-time Transfer Protocol (RTP) [SCF96].

Motivation for this work is to validate claims made by Sun for using Java and JMF to interface with a web cam and stream video over a network while interfacing with CORBA as a framework for building distributed systems that require a high degree of communication between autonomous applications running on separate personal computers. As a result, a video-based distributed monitoring system called Remote Object Webcam Controller (ROWC) was developed. ROWC uses a combination of these technologies to build a low-cost, scalable, cross-platform, distributed system to perform surveillance provided the major impetus to build the ROWC system.

Overview of Technology

Building a distributed system requires a great deal of network communication between distributed processes. Java applications may use sockets, RMI, or CORBA for communicating over a network. Although socket programming is efficient and has been widely used, it is also considered to be a lower-level form of network programming that is often tedious and error-prone, especially for large distributed systems. Sun's Remote Method Invocation (RMI) is a technology that provides a higher-level view of network programming. The underlying communication channel is made transparent to the programmer using RMI so method calls on a remote object are performed as if the object were local. RMI and CORBA are similar technologies in many respects, but, unlike CORBA, RMI requires an all-Java implementation [Sun02-2]. The choice to use CORBA over RMI is typically made from the need to interface with existing legacy systems or to avoid the restriction of being locked into a single programming language. Access to non-Java programs in RMI is possible using the Java native interface (JNI) [Sun02-5], but the overhead

required for using JNI makes CORBA an easier alternative. Although the ROWC system has been developed entirely in Java, CORBA was used to allow non-Java clients to be added to the system in the future and for future research integrating JMF with the CORBA Audio/Visual (A/V) service which is discussed later in this thesis.

The following sections provide a summary of the following technologies used to build the ROWC system: CORBA, video streaming, and JMF.

CORBA

The Object Management Group (OMG), a consortium of over 700 companies, has developed the Common Object Request Broker Architecture (CORBA) as a means to build interoperable software components that interact in a heterogeneous environment [OMG02]. Like RMI, CORBA greatly simplifies the programmer's work by providing a layer of abstraction between his application and the network programming layer. CORBA allows programmers to work within an object-oriented framework where remote objects are instantiated and their methods are called as if they resided on the same computer. Remote objects may be implemented in any high level language (HLL) like C++ or Java for which an Interface Definition Language (IDL) compiler is available. IDL is a declarative language developed by OMG for defining remote object implementations. By de-coupling the implementation language from its interface, neither the object nor the instantiator need to be concerned with the programming language that was used for each one's implementation.

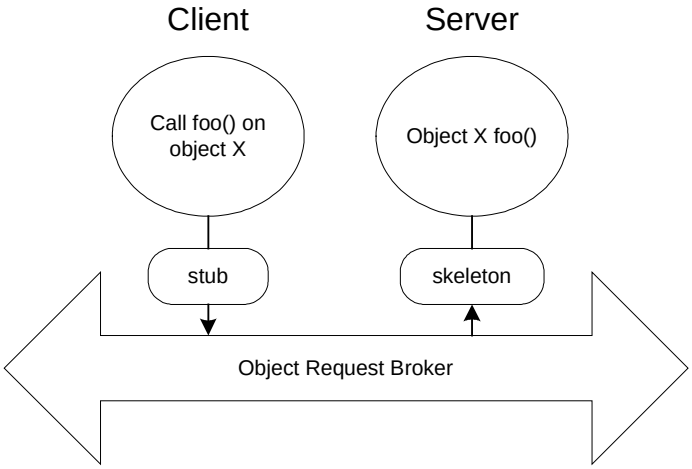

Figure 1 - CORBA client calling a method on object X.

A main component of CORBA is the Object Request Broker (ORB) which is responsible for marshalling method invocations, parameters, and return values across the network [Sca01]. All client requests are transparently sent to server object implementations by the ORB. ORB vendors that adhere to the CORBA standard are able to interact seamlessly. Figure 1 shows how the client invokes the method foo() on the object X that is housed on the server. The method call is marshaled through the client-side IDL stub to the server object. The server may return values back to the caller through the ORB. An IDL compiler generates both the IDL stub and skeleton.

The CORBA framework provides a number of services that are useful for building distributed applications. One of the most important services is the Naming Service. It allows objects to register themselves with the service so they can be made available to clients. The ORB uses the Naming Service to find remote objects and coordinate the marshaling of method calls and return values.

Video Streaming

Capturing and streaming live video has become increasingly popular on the Internet and local intranets. There are three methods for sending video over a network from a media server to a viewing client [Mic02-2]. In the first method, the complete file is downloaded from the server to the client. The client can then play the locally cached video file. This method is only acceptable when the video file is relatively small and the client does not need to see real-time video. With the second method, the video file is downloaded to the client, but the client is able to start viewing the video once enough of it has been cached; the video plays while it is downloading. This allows the client to start viewing the video sooner, especially when viewing large video files. This method is still not acceptable for real-time video since the reception of the video always lags behind the transmission. The third method sends video from the server to the client in real-time, and the client may or may not cache the video. This method requires significantly higher bandwidth for faster transmission than the previous two methods and is the only method acceptable for viewing live video. If the video is not cached, the server must re-transmit video for clients wanting a second viewing. This of course causes an increase in network traffic and increases the demands of the network. However, for viewing video in a pay-per-view scenario, this disadvantage becomes an advantage since free replay of cached content is not available. Because real-time video transmission requires immediate delivery, intermittent video frames must sometimes be dropped to cope with limited throughput. Although it is usually acceptable to lose frames periodically, a large delay in transmission of those frames is not acceptable. This method requires balancing the competing needs for a high quality picture and a high frame rate.

Streaming live video from a web cam requires the use of quick transport protocols like RTP [SCF96]. RTP is a higher layer transport protocol developed by the Internet Engineering Task Force (IETF) for providing end-to-end delivery of time-based media. It is typically used with the User Datagram Protocol (UDP) for its lower transport layer instead of the Transmission Control Protocol (TCP) [Mic02-2]. Connection-oriented transmission protocols like HTTP that are based on TCP are generally not acceptable for media transmission because of the additional overhead required for reliable connections such as retransmission of lost packets. IP delays in transmission due to packet retransmission can degrade live video. UDP does not guarantee that packets make it to their destination or arrive in the correct order; this is acceptable for video which can tolerate occasionally dropped packets [Wil02]. UDP is also connectionless which allows stream multicasting to multiple listeners. JMF provides hooks for using other transport protocols besides UDP. On a high-speed intranet where bandwidth is more plentiful, TCP or other native high-speed ATM protocols may be useful for their speed and Quality-of-Service (QoS) features.

Because RTP does not provide any mechanisms for ensuring transmission rate or quality, a control protocol (RTCP) is used with RTP to monitor the quality of data distribution and to monitor and identify RTP transmissions [Sun99]. Figure 2 shows the relationship of RTP to underlying transport protocols and media frameworks that use RTP.

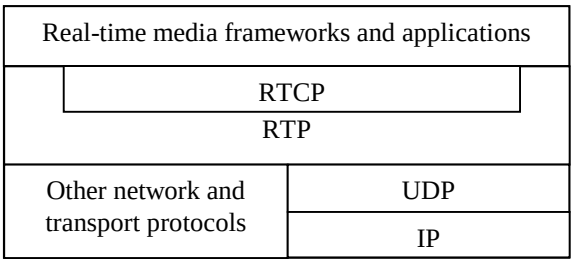

Figure 2 - RTP Architecture from [JMF99].

Java Media Framework

Java has long been touted for its platform independent benefits. Java Virtual Machines (JVM) are widely available for the most common operating systems. JMF is an application programming interface (API) for the development and incorporation of time-based media into Java applications that is easy to program and supports the streaming of video. JMF 2.1.1 supports SunVideo / SunVideoPlus capture devices on Solaris [Sun02-1] and almost any Windows capture device that support the Video For Windows (VFW) interface [Mic02-1]; JMF for Linux is supported through third-party software by Blackdown [Bla02].

JMF can be used by a Java application to interface with any JMF-compliant web cam. Video captured from the web cam is streamed using RTP although other transport protocol can also be used with JMF. JMF supports several video transmission formats including JPEG, H261, H263 MPEG-I. The ROWC system uses JPEG because it is fully supported on all platforms and uses the lowest RTP payload [Sup02]. JMF also supports a variety of media types like the popular Apple QuickTime [App02] format.

JMF uses a system of MediaLocators, SessionManagers, Players, Processors, DataSources, and DataSinks for coordinating the capturing, transmission, and reception of time-based media. These objects hide the underlying complexity associated with stream-based applications. These elements are briefly summarized here. A more detailed description can be found in the JMF 2.0 API Guide [Sun99].

- **MediaLocator** – identifier for a media DataSource, similar to a URL.
- **SessionManager** – coordinates RTP sessions by keeping track of the session participants and the streams that are being transmitted within the session.
- **Player** - processes an input stream of media data from a DataSource and renders it at a precise time. It provides standard user controls like play, pause, etc.
- **Processor** – specialized type of Player that provides greater processing control on an input media stream. It can output media to a media presentation device or to a DataSource.
- **DataSource** - delivers the input media-stream to a Player or Processor.
- **DataSink** - reads media data from a DataSource and renders the media to some destination (file, network, RTP broadcaster, etc.).

A MediaLocator object is used to access video for a web cam on a Windows computer like so:

```
MediaLocator mediaLocator = new MediaLocator("vfw://0");
```

Once the video feed is obtained, a DataSource is created to store the live video feed.

```
DataSource dataSource = Manager.createDataSource(mediaLocator );
```

This DataSource can be fed to a Player for the user to view the live video, or it can be sent to a Processor that transforms the DataSource into a different type of DataSource. The code segment below shows the web cam video feed being converted into an RTP-encoded DataSource for later transmission over the network.

```
Processor processor = Manager.createProcessor(dataSource);
// Transform into RTP DataSource (code is omitted for brevity)
DataSource rtpDataSource= processor.getDataOutput();
```

A converted DataSource can be stored to disk or transmitted across the network with the use of a DataSink. The following example shows how a DataSink would be used to transmit an RTP-encoded DataSource to the foobar computer on port 5050.

```
MediaLocator rtpLocator = new
      MediaLocator(“rtp://foobar:5050/video”);
DataSink rtpTransmitter = Manager.createDataSink(rtpDataSource,
```

```
        rtpLocator );
rtpTransmitter.open();
rtpTransmitter.start();
```

If the web cam's DataSource is to be transmitted over a network and recorded locally to file at the same time, it must be created as a cloneable DataSource. A cloneable DataSource can be cloned any number of times for different uses. The following shows how a cloneable DataSource is created from the web cam's DataSource:

```
DataSource cloneableDs =
      Manager.createCloneableDataSource(dataSource);
```

The JMF objects discussed in this section allow a Java application easy access to time-based media. A detailed description of how JMF is used by ROWC is discussed later in this thesis.

Architecture of System

The ROWC system is composed three applications: the Media Server, the Cam Processor, and the Cam Controller. All three of these applications were written in Java and use CORBA for communication. Each application may reside on the same computer or on separate computers in a LAN. ROWC could also be used over the Internet as long as each computer running the ROWC software is not hidden behind a firewall or other type of obstruction. Some small code modifications may also be required due to the pervasive use of computer host names which are unique on a LAN but aren't necessarily on the Internet. The unique host names are required by much of the software, and appending the host's domain name will be necessary to ensure the names are kept unique. For example, instead of just using the host name Oscar, the name Oscar.ualr.edu would be required.

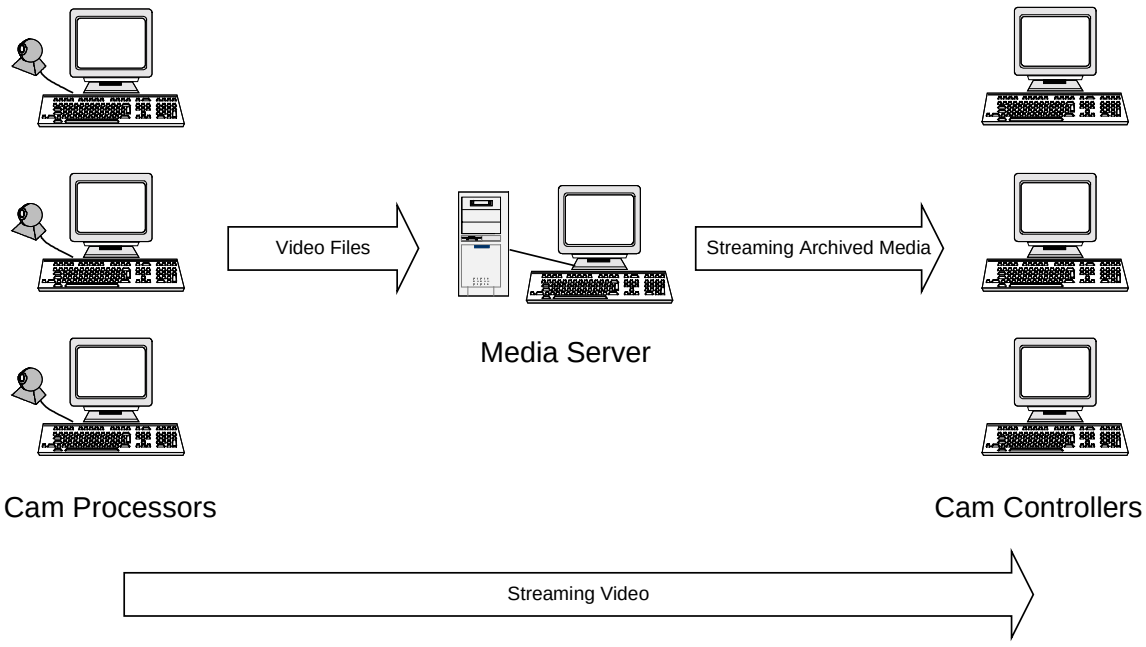

Figure 3 – ROWC overview.

Figure 3 shows a high-level view of ROWC. A web cam interfaces with a PC on which the Cam Processor application is executing. When a Cam Controller application operating on a PC decides to record video, the media/video file is created by the Cam Processor application and transmitted to the Media Server application for archiving. Any PC running the Cam Controller software can then view the media by having it streamed from the Media Server to the Cam Controller.

Detailed interaction of the Media Server, Cam Processor, and Cam Controller applications is shown in Figure 4. All three processes make use of distributed objects for interacting with each other. The CORBA Naming Service process is not displayed in Figure 4 for the sake of clarity. The Naming Service is for registering all CORBA objects so remote access to the objects is made transparent.

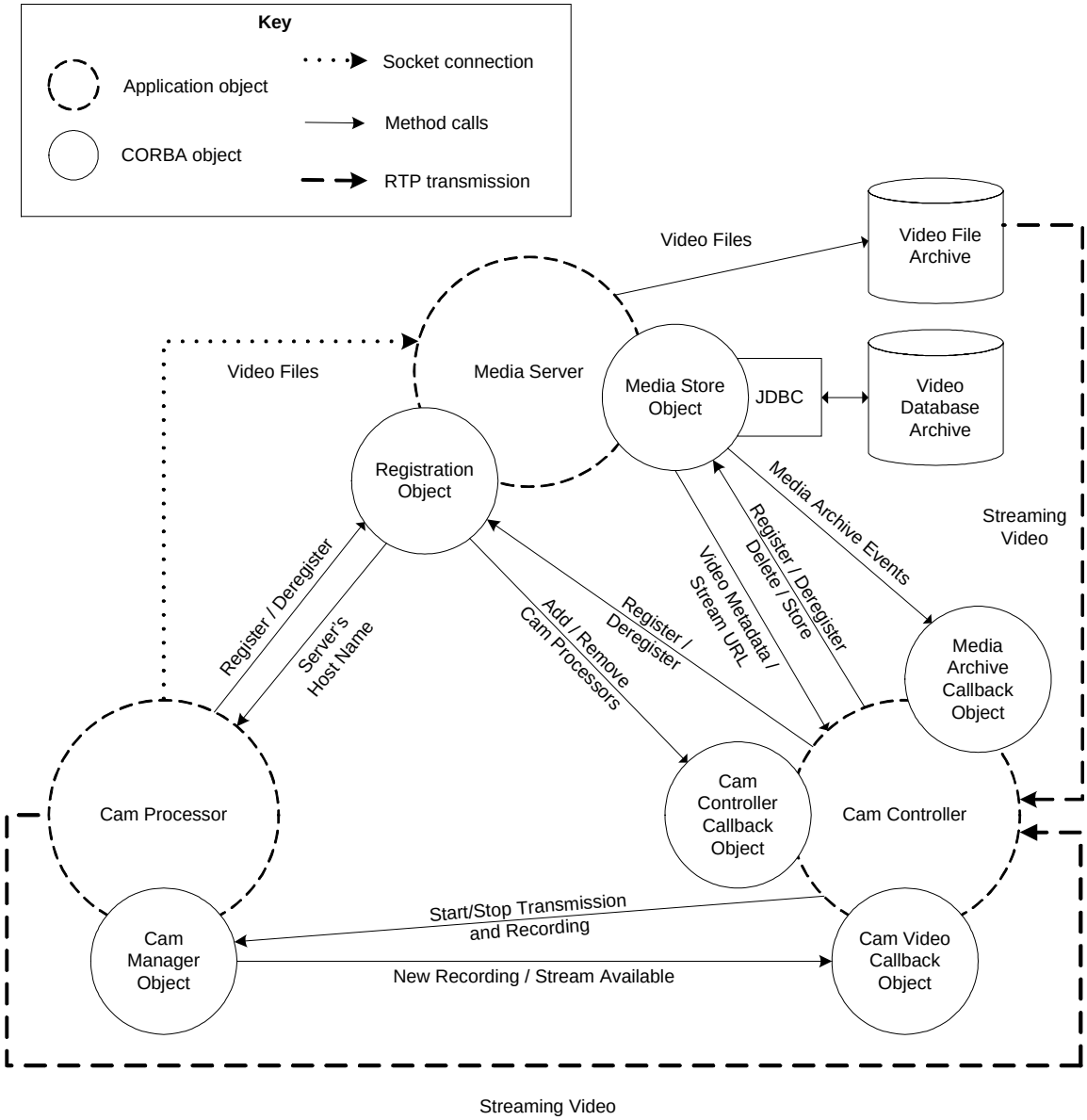

Figure 4 - Detailed view of ROWC.

ROWC is very scalable, allowing any number of Cam Processors and Cam Controllers to be added dynamically without changes to configuration files or source code. The system is ideally suited for an

intranet where high-speed access is available to networking resources due to the somewhat high bandwidth requirements of streaming video.

ROWC was built using IONA's ORBacus for Java 4.1 [ION02]. It is a free CORBA 2.4 compliant ORB. The Windows XP platform was used along with Java 2 version 1.3.1 for developing ROWC. The Logitech QuickCam [Log02] was used for surveillance on the PCs running the Cam Processor applications. The PC running the Media Server software used Microsoft SQLServer 2000 for storing recorded video.

The following sections will describe the major components of Figure 4 in detail. The sections will cover the Media Server, the Cam Processor, and the Cam Controller and the distributed objects they use for communication.

Media Server

The Media Server application is responsible for archiving video files obtained from Cam Processors and for streaming archived video to Cam Controllers. It is composed of the Registration object, the MediaStore object, and has a graphical user interface (GUI) for viewing registered Cam Processors, Cam Controllers, and archived media.

Figure 5 shows the Media Server application GUI. It displays a list of available Cam Processors and Cam Controllers and all the archived media for a selected location.

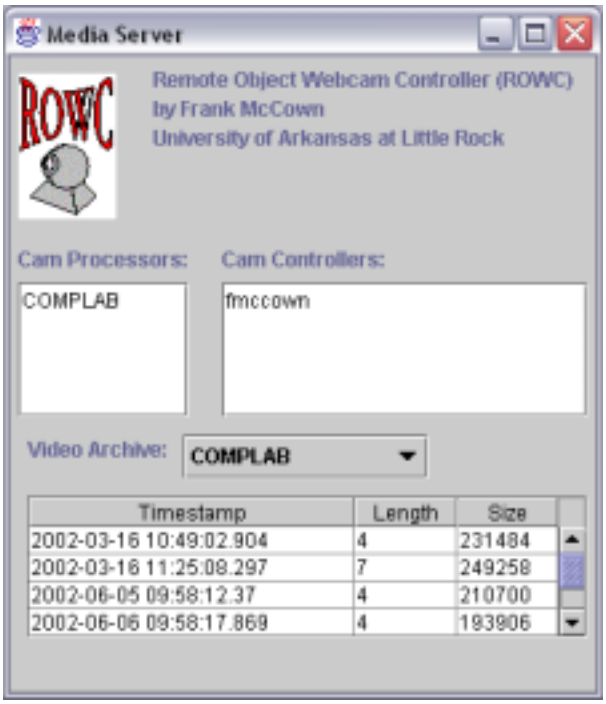

Figure 5 - Media Server application.

The Media Server must keep track of the media's originating location, timestamp, length (in minutes), and size (in bytes). This information is stored in the database, but the physical media file is left on disk. Because video files often take up large amounts of disk space, it is more efficient to leave them on disk than to place them into the database. Because all media files are stored in the same directory on disk, it is important to ensure that each file's name is unique. This is accomplished by combining the media file's location with its timestamp. Two media files from the same location should not have the same timestamp unless the computer the Cam Processor was running on was tampered with. For example, a video from the

COMPLAB location would be stored as COMPLAB_Fri_Jun_28_09_18_15_CDT_2002.mov. Any additional video files from COMPLAB would have a different timestamp and thus a different file name.

The Media Server runs in two separate threads. The first thread is for providing remote objects and handling CORBA requests. The second thread is a TCP-based server socket thread that listens for incoming video files from Cam Processors. Video files are sent to the Media Server using sockets instead of CORBA octet streams because most CORBA implementations have been shown to be inefficient when transporting large amounts of information due to excessive data-copying and expensive memory allocation per request [GS98]. Sockets provide the most efficient means for transporting large amounts of data. Unfortunately the high-performance benefits of socket programming are offset by its low-level interface that increases the development effort of building and extending the system. Researchers have introduced Blob Streaming Frameworks and other higher-level interfaces for gleaming the performance benefits provided by sockets while decoupling the underlying communication details [PHS96].

The Media Server implements two remote objects for use by Cam Processors and Cam Controllers: the Registration and MediaStore objects. The Registration object provides information necessary for Cam Processors and Cam Controllers to interact, and the MediaStore object is the Cam Controller's interface for accessing archived media. The following sections will give a detailed account of how the Registration and MediaStore objects are used.

Registration Object

The Registration object is housed by the Media Server for keeping track of available Cam Controllers and Cam Processors. Both Cam Processors and Cam Controllers must register with the Registration object when starting and stopping. Cam Controllers use the Registration object for discovery of available Cam Processors. Cam Processors use the object to determine the Media Server's host name so it that sockets can be used for uploading video files to the Media Server.

The IDL for the Registration interface is shown in Figure 6. A detailed discussion of this interface follows.

```
interface Registration
{
   // Notify MediaServer that a new CamController has started
   oneway void addCamController(in string hostName, in CamControllerCallback callback);

   // Notify MediaServer that CamController is going down
   oneway void removeCamController(in string hostName);

   // Notify MediaServer that a new CamProcessor is available
   oneway void addWebCam(in string location);

   // Notify MediaServer that a CamProcessor is no longer available
   oneway void removeWebCam(in string location);

   // Used by CamProcessor to get MediaServer's host name
   string getServerHostName();

   // Return array containing locations of all active CamProcessors
   StringArray getLocations();
};
```

Figure 6 - Registration object.

Most of the Registration object's methods are "oneway," meaning that a call to the method can return immediately and there is no expected return value from the method. This is ideal when a process does not need to wait for a method to complete its work. For example, the addCamController() method does not

need to wait for acknowledgement that it has indeed been registered with the Media Server. If the call did block, the Cam Controller would have to wait until the function call returned which would momentarily stop the Cam Controller from completing other necessary tasks.

The addCamController() method is used by a Cam Controller application to notify the Media Server of its existence. It is important that it registers with the Media Server so it will receive notification of the availability of various Cam Controllers that also register with the Media Server. The Cam Controller's host name is necessary for the Media Server to keep track of all the registered Cam Controllers. Since the host name is unique on a LAN, the Media Server can use it as a key. The CamControllerCallback is an object that is implemented on the Cam Controller and is therefore a remote object to the Media Server. It's methods are called by the Media Server when Cam Processors come up and down. See the Cam Controller section for further discussion of the CamControllerCallback object. When a Cam Controller is about to terminate, it uses the removeCamController() method to deregister with the Media Server so the Media Server will no longer attempt to send notifications to the Cam Controller.

The addWebCam() and removeWebCam() methods are used by Cam Processors to register and deregister with the Media Server. The Media Server keeps track of which Cam Processors are available so Cam Controllers will know what locations are available for viewing and recording. The location argument is a logical name for the Cam Processor's location like "COMPLAB" or "ROOM100".

The getServerHostName() method is used by a Cam Processor to determine the Media Server's host name on the network. The host name is used by the Cam Processor to open the socket connection to the Media Server so it can transport previously recorded video files to the Media Server.

The getLocations() method is used by a Cam Controller to determine what locations are available for controlling. All Cam Processors locations that have registered with the Media Server will be returned to the calling Cam Controller.

MediaStore Object

The Cam Processor uses the MediaStore object for video archiving and accessing information about the archive. The IDL for the MediaStore interface is shown in Figure 7.

```
interface MediaStore
{
   exception LocationsUnavailable{};
   typedef sequence<VideoClip> Clips;

   // Register for listening for media archive events
   oneway void addMediaArchiveListener(in string hostName,
                                       in MediaArchiveCallback callback);

   // Deregister for media archive events
   oneway void removeMediaArchiveListener(in string hostName);

   // Returns array of web cam names from the database.  Raise the LocationsUnavailable
   // exception if there is an error retrieving locations.
   StringArray getArchiveLocations() raises(LocationsUnavailable);

   // Return information from database about all archived media at a given location
   void getAllVideoDetails(in string location, out Clips videos);

   // Deletes a set of videos for a given location from the database.  Return true if successful.
   boolean deleteVideos(in string location, in StringArray names);

   // Store a video in the database.
   oneway void storeVideo(in VideoClip videoRec);
```

```
    // Start streaming a particular video via RTP to the given destination IP address.
    string getVideoStream(in string location, in string name, in string destIPaddr);
};
```

Figure 7 - MediaStore object.

All Cam Controllers register themselves for receiving notification of changes to the archive media library using addMediaARchiveListener() when first initializing. When a new video is added to the archive or a video is deleted, all Cam Controllers need to update their interface to display the change. Otherwise a user with an outdated list of archived videos may attempt to view a video which had previously been deleted by another Cam Controller user. When a Cam Controller is going down, the removeMediaArchiveListerner() method is used to de-register for media archive events.

The getArchiveLocations() method is used by Cam Controllers to determine what media archive locations are available. (Throughout this paper, web cam *locations* and *names* will be synonymous. The web cam name by default is the Cam Processor's host name, but a more descriptive location name can also be provided at startup time.) The archive location list is presented to the user for browsing the archive library. The getAllVideoDetails() method provides the Cam Controller with timestamp, length, and size information about the stored media. If the user of a Cam Controller wishes to delete a video, the deleteVideo() method is called to remove the media from the database and from the Media Server's file system.

The JDBC API is used for interfacing to the relational database system because it provides a standard access to a wide variety of databases through the use of Structured Query Language (SQL). The ROWC system was developed using Microsoft SQLServer 2000, but any relational database for which a JDBC driver exists could be used.

When a Cam Controller has finished recording video for a Cam Processor and transmitted the video via sockets to the Media Server, the Cam Processor calls the storeVideo() method to send notification to the Media Server that the previously transmitted video should be stored in the database. The physical video file is also renamed to match its name in the database.

Cam Processor

The Cam Processor application can be used on any personal computer that has a JMF compliant web cam attached to it. Figure 8 shows the Cam Processor application GUI. It displays live feed from the web cam and allows the user to stop viewing the live video if so desired.

Figure 8 - Cam Processor application.

The Cam Processor application houses the CamManager object for use by any Cam Controller wanting to control the Cam Processor's web cam. The CamManager allows a Cam Controller to receive live streaming video and to record video that is archived on the Media Server. A callback mechanism is used to notify the Cam Controller when a video stream is available or when a video has completed recording.

The IDL for the CamManager interface is shown in Figure 9.

```
interface CamManager
{
   // Start streaming video to given host name.  Callback's videoAvailable called when video
   // starts being transmitted.  Returns false if video can't be transmitted because CamProcessor
   // is busy transmitting elsewhere, true otherwise.
   boolean startTransmission(in string hostName, in CamVideoCallback callback);

   // Stop current transmission of video.
   oneway void stopTransmission();

   // Start recording video from web cam to file for a maximum of milliseconds.  The callback
   // is used to notify the CamController when the recording has been completed.
   boolean startRec(in long milliseconds, in CamVideoCallback callback);

   // Stop recording video.
   oneway void stopRec();
};
```

Figure 9 - CamManager object.

The startTransmission() method is called when the Cam Controller's user wants to view live video from the selected Cam Processor. The Cam Controller's host name is sent to startTransmission() so that the Cam Processor knows where to transmit the video. The CamVideoCallback object is used by the Cam Processor for notifying the Cam Controller when the streaming video is available for viewing via RTP; initialization of JMF takes a few seconds before transmission of the video begins. When the Cam Controller's user is no longer wanting to see live video from the web cam, the stopTransmission() method is used to stop the streaming of the live video.

The startRec() method is used by the Cam Controller to begin recording of live video from the selected web cam to file. The recording of video will stop when the maximum number of milliseconds has been reached or when the stopRec() method is called. The QuickTime format is used for storing the video file on the Cam Processor's hard drive. The CamVideoCallback object is use by the Cam Processor to notify the Cam Controller when the recorded video is available for viewing. The video is only available when it has been uploaded to the Media Server. Video can be streamed from the Cam Processor to the Cam Controller at the same time it is being recorded. Unfortunately the extra processing load on the CPU tends to degrade the quality of both the transmitted and recorded video.

Another method that could have been employed for transmitting the recorded video to the Media Server would have been to send an RTP stream during recording to the Media Server instead of recording the video to file on the Cam Processor's computer. The Media Server could then store the receiving RTP stream to file, and uploading from the Cam Processor would no longer be required. Unfortunately this solution would have resulted in degraded video quality due to lost frames during RTP transmission. Recording the video directly to the Cam Processor's hard drive results in the highest quality of video.

Because Cam Processors do not make use of the MediaStore remote object, they do not have direct access to the Media Server's archive, nor do they need it. When a Cam Processor uploads a file to the Media Server using a socket connection, the file is not immediately stored in the database. It remains unprocessed on the Media Server's hard drive until the Cam Controller notifies the Media Server of the new media using the MediaStore object. Once notified, the Media Server then moves the uploaded video file to its archive directory and stores the video metadata in the database.

Cam Controller

The Cam Controller application allows the user to control any Cam Processor's web cam remotely. Figure 10 shows the Cam Controller application GUI. The user may select any available location for viewing, can start and stop recording, change the maximum recording time, open a local media file for viewing, and can view or delete any archived media from the selected location.

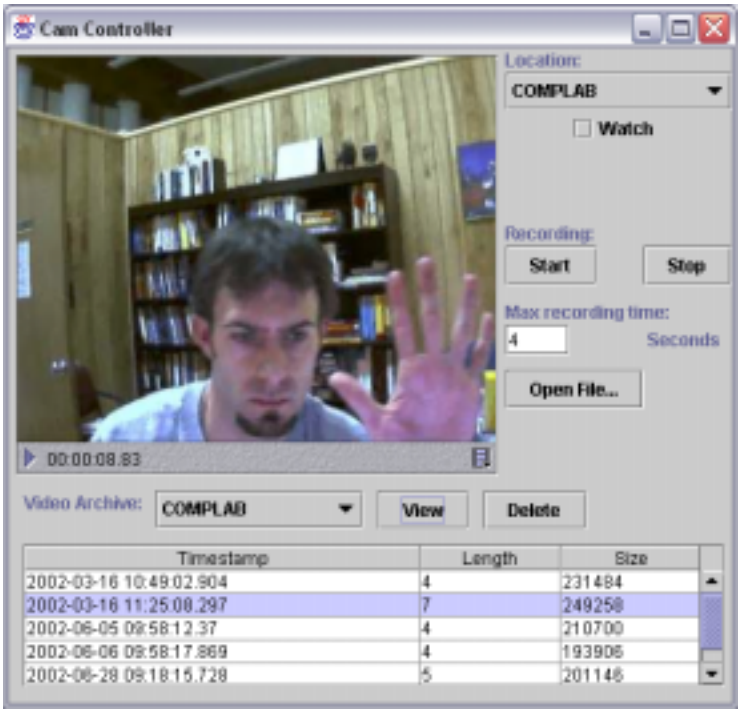

Figure 10 - Cam Controller application.

The Cam Controller makes heavy use of the MediaStore object for accessing archived videos and the CamManager object for controlling web cams remotely. It only houses three objects: the CamVideoCallback object for notification of live video streams and recordings, the CamControllerCallback object for Cam Processor availability notification, and the MediaArchiveCallback object for notification of new and deleted archive media. These objects are only used for callback notification of events and are therefore always passed via a remote method invocation to a remote object. The remote object invokes the proper method on the given object in order to notify the owner of the object of a certain event. The callback model is explained in more detail in the "Dynamic View Updating of Media Archive Library" section.

Figure 11 shows the CamVideoCallback, CamControllerCallback, and MediaArchiveCallback IDL interfaces.

```
interface CamVideoCallback
{
    // Notify CamController of a new video clip that has been recorded.
     oneway void newRecording(in VideoClip video);

     // Notify CamController of video that is being streamed to the given URL.
     oneway void videoAvailable(in string url);
};
```

```
interface CamControllerCallback
{
    // Notifies all CamControllers that a new CamProcessor is available
    oneway void addWebCam(in string location);

    // Notifies all CamControllers that a CamProcessor is no longer available
    oneway void removeWebCam(in string location);
};

interface MediaArchiveCallback
{
    // Notify CamProcessor that a video has been archived for this location
    oneway void addVideo(in string location);

    // Notify CamProcessor that a video has been deleted at this location
    oneway void removeVideo(in string location);
};
```

Figure 11 – CamVideoCallback, CamControllerCallback, and MediaArchiveCallback objects.

The CamVideoCallback object's newRecording() method is called by a Cam Processor when recording of a video from a web cam has completed and been uploaded to the Media Server. It notifies the Cam Controller of the details of the video such as length, time stamp, and size (all members of the VideoClip structure).

The videoAvailable() method is used by a Cam Processor to notify the Cam Controller that streaming video is available for viewing. In order to view the streaming video, the Cam Controller only needs the URL which contains the port number on which the Cam Processor is transmitting the video to the Cam Controller. For example, the following URL would be transmitting video on port 5010 to the foobar computer: `rtp://foobar:5010/video`

The CamControllerCallback object is used by a Cam Controller to be notified by the Media Server when a location is/is not available for viewing. When a Cam Processor starts up, it registers with the Media Server, and the Media Server uses each Cam Controller's CamControllerCallback object to notify it of the newly available location. Similarly, when a Cam Processor goes down, the Media Server notifies all the Cam Controller's of the Cam Processor's demise.

The Media Server's MediaStore object uses a Cam Controller's MediaArchiveCallback object to notify the Cam Controller of additions and deletions to the media archive by calling the MediaArchiveCallback's addVideo() and removeVideo() methods. The Cam Controller needs this information so it can update it GUI representation of the media archive.

Dynamic View Updating of Media Archive Library

Both the Media Server and the Cam Controller need to update their GUI when changes to the underlying media archive library occur. The model-view-controller (MVC) design pattern [GHJ95] was used to efficiently implement this problem. The MVC design pattern consists of three entities that are logically separated to enhance the ability to modify and add-on to the application: the model, the controller, and the view. The model represents the application data, the media archive library. The controller handles all changes to the model, the receiving of new videos or deletions to the archive. The view represents the outward appearance of the model, the table showing all the media archive data. So whenever the MediaStore object (the model) is changed, the Media Server and Cam Controllers are notified to update their GUI (the view).

Figure 12 shows two different methods for dynamically updating each view since the MediaStore object is local for the Media Server and remote for the Cam Controller. On the Media Server, the events

are implemented using the same event-handling mechanism that Java Swing and AWT components use [DD02]. The Swing/AWT event-handling mechanism involves providing an event listener interface which is implemented by an event handler. The event handler is registered for notification of the event by the listening object, and the event handler is then notified when the particular event occurs. Using this event-handling mechanism, the Media Server registers as a MediaArchiveListener event handler which receives MediaArchiveEvents when new videos are received or videos are deleted. The events are then reflected in the GUI.

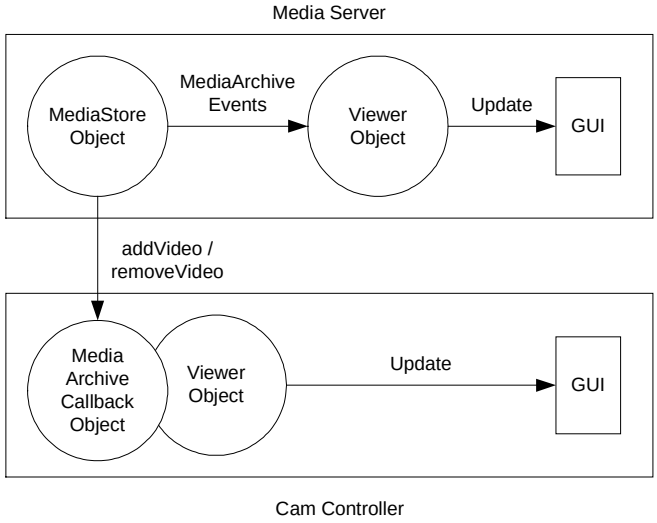

Figure 12 - Updating GUI when media archive is changed.

Cam Controllers on the other hand use the CORBA callback model [TS02]. The callback model is useful for asynchronous event notification. For example, assume a client needs to know about a particular event that will eventually take place on the server. The client will need to provide an object to the server that contains methods the server will invoke when the events take place. Because the client's object is remote to the server, it is defined with IDL just like any other CORBA object. Using the callback model, each Cam Controller registers itself for media archive events using a locally implemented MediaArchiveCallback (Figure 11) object. The object is sent to the MediaStore when the Cam Controller initializes. The MediaStore then calls the MediaArchiveCallback's addVideo() and removeVideo() methods when changes to the media archive are made.

OMG's CORBA Event service [OMG01] or the newer Notification service [OMG00-1] could also be used for notifying the Cam Controller of changes to the MediaStore. The Event service provides a mechanism for pushing or pulling un-typed events from an event supplier to an event consumer. An event channel decouples the consumer from the supplier. In the push model, the supplier pushes an event through the event channel which is then passed on to the consumer. In the pull model, the consumer polls the event channel for new events from the supplier. The Notification service builds upon the Event services by providing the ability to register for particular types of events. In the ROWC scenario, Cam Controllers would register themselves as event consumers, and the Media Store would register itself as an event producer. Using the push model, the Cam Controllers would be notified asynchronously when a change to the media archive was made. The CORBA event services are useful in larger distributed systems where management and proliferation of events are more difficult to control.

ROWC JMF Architecture

This section provides a summary of how JMF is used by ROWC to capture video from a web cam, display live video to a user, transmit live video across a network, and store live video to disk.

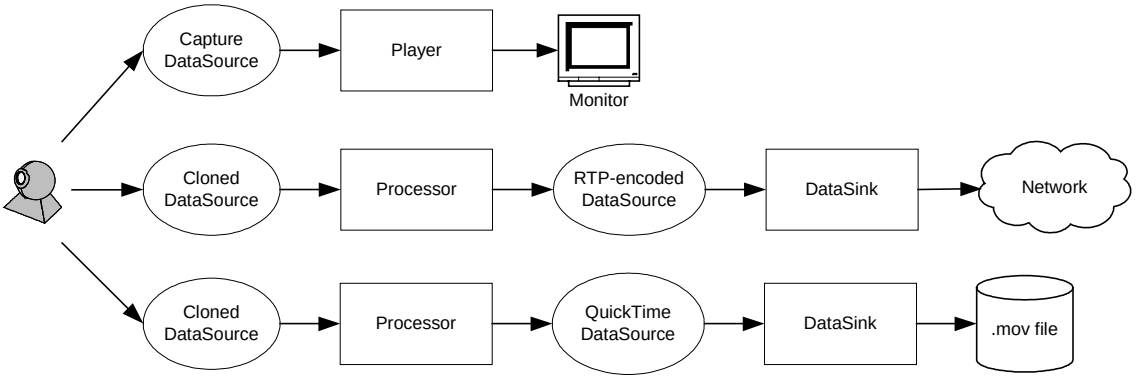

Figure 13 - Cam Processor: Video from a web cam is captured and cloned for viewing, streaming, and storing to file.

Figure 13 shows how the Cam Processor application clones the DataSource that is associated with the web cam. A Player is used to view the live feed from the web cam, and separate Processors are used to transmit the video across the network and record to file. The QuickTime format is used for storing video to file.

Figure 14 shows how a Cam Controller application displays streaming video from a Cam Processor or Media Server to the user. The streaming video is accessed using an RTP SessionManager. When a video stream becomes available, the SessionManager creates an RTP-encoded DataSource that is fed into a Player. The Player allows the Cam Controller user to see the live video stream. If the streaming video is halted because the Cam Processor is brought down or the video file has finished being streamed from the Media Server, the SessionManager is alerted and stops the Player. A SessionManager would not be necessary if the Media Server notified the Cam Controller via callbacks when the video streaming had finished. In the case of the Cam Processor, the SessionManager is not necessary at all since the Media Server already notifies all Cam Controllers when a Cam Processor is no longer available. Figure 14 shows this process in detail.

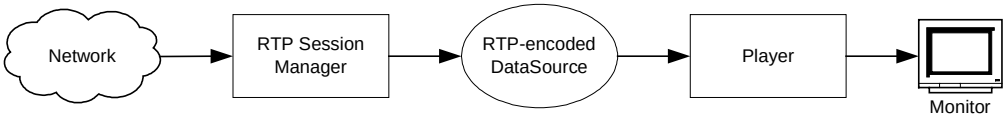

Figure 14 – Cam Controller: Streaming RTP video is sent to a Player for viewing.

Figure 15 shows how the Media Server application transmits archived video over the network to Cam Controllers. QuickTime video files are first converted to JPEG/RTP encoding. Then it is transmitted over the network using a DataSink. Once the video has finished streaming, the DataSink and Processor are closed and removed. Closing of the RTP channel signals the receiving Cam Controller that transmission of the archived media is complete.

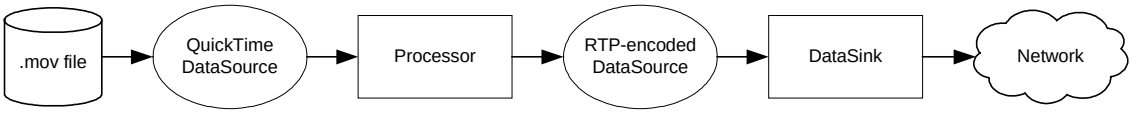

Figure 15 – Media Server: A QuickTime file is streamed with RTP over the network.

Related Work

The ROWC system concept of remote video access could be applied to many domains such as teleconferencing, virtual collaboration, and telepresence.

Although JMF provides the capability to interface with web cams and to stream time-based media, it does not currently provide a higher-level API for QoS provisioning, security, and media archiving. These components must be built on top of JMF. The use of multimedia middleware can provide these missing components without re-implementation.

MAESTRO [HSK97] and Da CaPo++ [SCW99] are two recent implementations of multimedia middleware that have been developed to support the development of distributed multimedia communication applications. Both sets of middleware provide a layer between a particular multimedia device API and the underlying network used to transmit the multimedia. The middleware is not intended to fill the role that JMF which provides device-independent access to multimedia devices like web cams. They instead offer an array of services not provided by JMF that facilitate the building of distributed multimedia systems. The services include session management (for managing the connection between media producers and consumers), security services (for encryption and authentication of streams, users, and QoS parameters), and QoS management (for dynamic changes to QoS). MAESTRO provides an additional media storage service that is useful for storing multimedia objects transparently.

OMG offers an audio/video (A/V) streaming model for CORBA-based distributed multimedia streaming frameworks [OMG00-2]. The model allows stream connection and management with higher-level CORBA operations while using efficient lower-level transport protocols like ATM, UDP, TCP, and RTP. The A/V Streaming Service has been implemented by the ACE ORB (TAO) [ACE02] and demonstrated [LH00] to be useful for hiding the complexities of stream transport.

Another approach to the A/V streaming model in CORBA is taken by MULTE-ORB [EKP00]. MULTE-ORB is an attempt to provide additional multimedia middleware capabilities to an ORB by providing stream binding and management for stream flows within the CORBA paradigm. It also provides a flexible protocol framework that allows transport protocols to be dynamically modified or reconfigured.

DSS Surveillance System Implementation

ROWC shares several characteristics with the Distributed Surveillance Services (DSS) system developed at the National Kaohsiung First University [LC99]. DSS was built using Java/CORBA/RMI with a C++ interface for the video camera. JMF was not used for implementation because of its lack of maturity at the time DSS was built. Thus the DSS video camera interface is inherently non-portable without code changes.

In order to reduce network load, DSS stores a series of JPEG images instead of a video stream. Each image is placed in an array of bytes for transferring across the ORB. This approach was also used in the first version of the ROWC system when only a single snapshot could be captured from a web cam. Since a JPEG image is relatively small compared to a video stream, using the ORB for the transport layer was convenient and efficient. This approach was later abandoned when ROWC was converted from still images to video.

DSS used signed Java applets for deployment of Surveillance Monitors, similar to Cam Controllers. This approach requires downloading ORB support files along with the applet which can make accessing the application rather slow initially. Developers benefit from implementing the system with applets because it is easier to deploy the system (the user only needs a Java-enabled browser) and bug fixes can quickly be dispatched (each time the applet is accessed for the first time, the most recent version of the software is downloaded). Similar benefits are also provided by Sun's newest model of application deployment called Java Web Start (JWS) [Sun02-3]. JWS provides quicker launching of applications than using the applet model because all application files are cached on the user's machine. Applets only remain cached by the browser for a limited time. ROWC could be distributed with JWS with no code changes.

ezlinX Surveillance System

A successful commercial video surveillance system called ezlinX was recently built using JMF to provide security for small to medium-sized businesses [NOO02]. ezlinX provides live video transmission and recording like ROWC but also provides a host of other features like authentication, motion detection, alarms, etc. Although JMF was used for building their software, they had to re-write several major components of JMF due to limitations of the framework [Bin02].

Future System Enhancements

There are several directions that ROWC could be taken for future research and improvements that would make it more useful as a true surveillance system. A summary of these items is presented here.

QoS Provision

A provision for QoS constraints on a ROWC video stream would produce higher quality video for Cam Controllers that operate in heterogeneous environments. For example, a Cam Controller operating on a low bandwidth network would need a video streamed with a lower frame rate in order to see live video from a Cam Processor. A Cam Controller operating on a higher bandwidth network could receive the same video stream at a higher frame rate.

A video conferencing system using JMF called JQoS demonstrates the successful use of QoS measures using an intermediary SessionManager between a Source and Receiver process [ZG01]. The SessionManager regulates the frame rate between Source and Receiver by monitoring RTCP reports and QoS requests from the Receiver and making the necessary QoS adjustments. Some adjustment requests are forwarded to the Source which may also need to adjust its transmission to match the QoS parameters. A similar integration of QoS constraints into ROWC would be especially useful if ROWC were to be used outside a local intranet where network heterogeneity is the norm. Earlier discussions in this paper mentioned the use of multimedia middleware that also provides QoS services.

Multiple Access of Cam Processor

ROWC only allows a single Cam Controller to view streaming video from a single Cam Processor. If a second Cam Controller tries to view video from a busy Cam Processor, it is denied access. If this functionality were necessary, the Cam Processor would need to use an RTP SessionManager to clone additional web cam DataSources for streaming to multiple Cam Controllers [Sun99]. This would degrade video quality for each Cam Controller due to the overhead incurred when streaming video for each session. For transmission to more than a few Cam Controllers, a better approach would be to use a multicast transmission. This would require some additional overhead to organize Cam Controllers into members of a specified multicast group for receiving video from the same Cam Processor. Multicast transmissions are usually not supported outside of local intranets.

Media Server Improvements

ROWC is ideal in a small environment where concurrent access to the Media Server is limited to one or two Cam Controllers at a time. Because the Media Server stores all video files on its local hard disk, a large number of concurrent accesses of the same video file by multiple Cam Controllers could lead to degraded transmission. Increasing the number of concurrent accesses can be improved with file replication (making multiple copies of the same file on different disks) or data stripping [Gem95]. The data stripping

method stores a single video file across multiple disks so that the file can be read in parallel. This method takes less disk space than file replication, but requires increased complexity in file storage.

A method for providing fault tolerance to ROWC is crucial to keep archived media from disappearing in the event of a hard drive crash or network interruptions incurred by the Media Server. In the first case, redundancy of video files and database information is necessary for the Media Server to retain an up-to-date archive. Mirroring schemes have shown to be useful in improving fault tolerance of media servers [Mou96] although the increase in storage volume is somewhat costly.

Conclusion

The ROWC system was built using Java, JMF, and CORBA to perform surveillance using a personal computer and JMF-enabled web cam. The ability to archive recorded video was also built into ROWC. The three applications making up ROWC (the Media Server, Cam Processor, and Cam Controller) can be ran on separate computers connected in a network or on the same PC.

The combination of CORBA and JMF has been demonstrated by the ROWC system to be sufficient for producing a distributed video surveillance system that is portable, flexible, and extendable. Many areas of ROWC are open for improvement, such as provision for QoS constraints and providing a multicast stream from a single Cam Processor's web cam. Related work demonstrating the use of multimedia middleware and standardized CORBA media interfaces could also be used to improve ROWC.

Our experience in building ROWC has uncovered some JMF bugs that have caused core dumps. Other developers have experienced similar problems with JMF [JMF02-1]. Sun is currently in the process of refining and extending JMF, and future versions are likely to be more stable [JMF02-2]. Sun also provides the complete JMF source code for developers who need to fix bugs, extend functionality, or remove bottlenecks for use in their application.

Appendix

A representative amount of code from the ROWC system is shown here. For access to the complete source code, please contact the author.

// **rowc.idl**

```
// IDL interface definitions for ROWC system.

//  Copyright (C) 2002 by Frank McCown
//  University of Arkansas at Little Rock
//  July 2002

// Compile with Orbacus: jidl rowc.idl


module rowc
{
  // Unbounded "array" of strings
  typedef sequence<string> StringArray;


  struct VideoClip
  {
```

```
  string location;       // Location of the video
  string name;           // Video name determined by timestamp
  long size;             // Size of video in bytes
  long length;           // Length in seconds
  long long timeStamp;   // In milliseconds
 };



 interface CamVideoCallback
 {
  // Notify CamController of a new video clip that has been recorded.
  oneway void newRecording(in VideoClip video);

  // Notify CamController of video that is being streamed to the given URL.
  oneway void videoAvailable(in string url);
 };

 interface CamManager
 {
  // Start streaming video to given host name.  Callback's videoAvailable called when video
        // starts being transmitted.  Returns false if video can't be transmitted because CamProcessor
        // is busy transmitting elsewhere, true otherwise.
  boolean startTransmission(in string hostName, in CamVideoCallback callback);

  // Stop current transmission of video.
  oneway void stopTransmission();

  // Start recording video from web cam to file for a maximum of milliseconds.  The callback
  // is used to notify the CamController when the recording has been completed.
  boolean startRec(in long milliseconds, in CamVideoCallback callback);

  // Stop recording video.
  oneway void stopRec();
};

interface MediaArchiveCallback
{
    // Notify CamProcessor that a video has been archived for this location
  oneway void addVideo(in string location);

  // Notify CamProcessor that a video has been deleted at this location
  oneway void removeVideo(in string location);
};

interface MediaStore
{
  exception LocationsUnavailable{};
  typedef sequence<VideoClip> Clips;

    // Register for listening for media archive events
  oneway void addMediaArchiveListener(in string hostName, in MediaArchiveCallback callback);

  // Deregister for media archive events
  oneway void removeMediaArchiveListener(in string hostName);
```

```
    // Return unbounded array of web cam names from the database.
    // Raise LocationsUnavailable if error retrieving locations.
    StringArray getArchiveLocations() raises(LocationsUnavailable);

    // Return information from database about all archived media at a given location
    void getAllVideoDetails(in string location, out Clips videos);

    // Deletes a set of videos for a location from the database.  Return true if successful, false otherwise.
    boolean deleteVideos(in string location, in StringArray names);

    // Store a video in the database.
    oneway void storeVideo(in VideoClip videoRec);

    // Start streaming a particular video via RTP to the given destination IP address.
    string getVideoStream(in string location, in string name, in string destIPaddr);
};


interface CamControllerCallback
{
     // Notifies all CamControllers that a new CamProcessor is available
    oneway void addWebCam(in string location);

    // Notifies all CamControllers that a CamProcessor is no longer available
    oneway void removeWebCam(in string location);
};

  interface Registration
  {
     // Notify MediaServer that a new CamController has started
     oneway void addCamController(in string hostName, in CamControllerCallback callback);

    // Notify MediaServer that CamController is going down
    oneway void removeCamController(in string hostName);

    // Notify MediaServer that a new CamProcessor is available
    oneway void addWebCam(in string location);

    // Notify MediaServer that a CamProcessor is no longer available
    oneway void removeWebCam(in string location);

    // Used by CamProcessor to get MediaServer's host name
    string getServerHostName();

    // Return array containing locations of all active CamProcessors
    StringArray getLocations();
  };
};
```

```
/*   CamProcessor.java
 *
 *   Application that interfaces with a web cam and allows remote CamControllers
 *  to control the web cam.
 *
 *   Optional startup arguments:
```

```
 *   -loc  used for overriding location name.  Default is computer's host name.
 *   -file used to play a local QuickTime movie file
 *
 *  Copyright (C) 2002 by Frank McCown
 *  University of Arkansas at Little Rock
 *  July 2002
 */

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.PortableServer.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;
import javax.media.*;
import javax.media.protocol.*;
import rowc.*;


public class CamProcessor extends JFrame
{
    private final int DEFAULT_WINDOW_WIDTH = 470;
    private final int DEFAULT_WINDOW_HEIGHT = 370;

    private static final String PROPERTY_FILE = "project.conf";

    private static String fileSpecified = null;  // Used for movie files (when no web cam is available)

    public static boolean quitting = false;

    private static NameComponent camManagerNC[];
    private static NamingContext ncRef;

    public static Registration regObject;       // Ref to remote object
    private static CamManagerImpl camManager;   // Created servant

    private JButton btnGetFile;
    public static JLabel lblInfo;
    private JLabel lblLocation;
    private JCheckBox viewVideoCheckBox;
    public static JScrollPane videoScrollPane;
    public static VideoTransmitter vt;
    public static TransmitterThread transThread;
    private ProcessorVideoPanel videoPanel;        // Panel that displays live video
    public static DataSource dataSource = null;    // Connected to web cam

    public static String location = "";  // Unique identifier for cam processor

    // Constructor
    private CamProcessor()
    {
        super("Cam Processor");

        videoPanel = new ProcessorVideoPanel();
```

```
    // Tell Server about us so he'll add us to his list
    if (regObject != null)
        regObject.addWebCam(location);

    // Start thread to produce video for video player
    ViewingThread vThread = new ViewingThread();
    vThread.start();

    Container c = getContentPane();
    c.setLayout(new GridBagLayout());

    videoScrollPane = new JScrollPane(videoPanel);

    //for loading two frames
    viewVideoCheckBox = new JCheckBox("View Video", true);
    viewVideoCheckBox.addItemListener(new ItemListener()
    {
        public void itemStateChanged(ItemEvent e)
        {
            // Show/hide video panel when selected/deselected

            if (e.getStateChange() == ItemEvent.SELECTED)
                videoPanel.setVisible(true);
            else if (e.getStateChange() == ItemEvent.DESELECTED)
                videoPanel.setVisible(false);
        }
    });

    lblLocation = new JLabel("Location: " + location);

    lblInfo = new JLabel();
    lblInfo.setForeground(Color.red);

    c.add(videoScrollPane, new GridBagConstraints(0, 0, 1, 1, 1.0, 1.0,
        GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(5, 5, 5, 5), 217, 202));
    c.add(viewVideoCheckBox, new GridBagConstraints(1, 0, 1, 1, 0.0, 0.0,
        GridBagConstraints.CENTER, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
    c.add(lblLocation, new GridBagConstraints(0, 1, 2, 1, 0.0, 0.0,
        GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
    c.add(lblInfo, new GridBagConstraints(0, 2, 2, 1, 0.0, 0.0,
        GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));


    setSize(DEFAULT_WINDOW_WIDTH, DEFAULT_WINDOW_HEIGHT);
    show();
}


private void showError(String error)
{
    JOptionPane.showMessageDialog(this, error, "CamProcessor Error",
        JOptionPane.ERROR_MESSAGE);
}

private static void activateCamManager(org.omg.CORBA.ORB orb)
{
```

```
    try
    {
        // get reference to rootpoa & activate the POAManager
        POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
        rootPOA.the_POAManager().activate();

        // Create the servant
        camManager = new CamManagerImpl(location);
        rowc.CamManager href = camManager._this(orb);

        String host = regObject.getServerHostName();
        camManager.setServerName(host);

       // Get the root naming context
        ncRef = NamingContextHelper.narrow(
        orb.resolve_initial_references("NameService"));

        // Bind the Object Reference in Naming
        String camName = "CamManager_" + location;
        camManagerNC = new NameComponent[1];
        camManagerNC[0] = new NameComponent(camName,"");
        ncRef.rebind(camManagerNC, href);
    }
    catch (Exception e)
    {
        System.out.println("Error trying to activate CamManager implementation.");
        e.printStackTrace();
    }
}

public static void transmitVideo(String hostName)
{
    // Create new thread for transmitter going to new location
    transThread = new TransmitterThread(hostName);
    Thread thisThread = Thread.currentThread();
    transThread.start();
}

public static void stopTransmitVideo()
{
    transThread = null;
}

 public static void main(String[] args)
{
    // Attempt to read location from command line

    for (int i=0; i < args.length; i++)
    {
        if (args[i].equals("-loc"))
        {
            if(i+1 >= args.length)
            {
                System.err.println("Argument expected for " + args[i]);
                System.exit(0);
            }
```

```
            StringBuffer loc = new StringBuffer(args[i+1]);
            location = loc.toString().toUpperCase();
            i++;  // Skip "-loc" next iteration
        }
        else if (args[i].equals("-file"))
        {
            if(i+1 >= args.length)
            {
                System.err.println("Argument expected for " + args[i]);
                System.exit(0);
            }
            fileSpecified = new StringBuffer(args[i+1]).toString();
            i++;  // Skip "-file" next iteration
        }
    }

    // Get ip addr for this computer to be used for creating a receiver
    try
    {
        java.net.InetAddress ia = java.net.InetAddress.getLocalHost();
        String ipAddr = ia.getHostAddress();
        if (location.length() == 0)
            location = ia.getHostName().toUpperCase();   // Use host name for location
    }
    catch (java.net.UnknownHostException ex)
    {
        ex.printStackTrace();
    }

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");

    try
    {
        // Read properties from project property file that indicates location
        // of Name and Event Services.

        FileInputStream f = new FileInputStream(PROPERTY_FILE);
        props.load(f);
        f.close();
    }
    catch (Exception e)
    {
        System.out.println("Could not load properties from " + PROPERTY_FILE);
        e.printStackTrace();
    }

    final org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);

    // Get objects by name
    try
    {
        // Get naming service
        NamingContext nc = NamingContextHelper.narrow(
            orb.resolve_initial_references("NameService"));
```

```
            // Get remote object using Naming Service
            NameComponent[] ncArray = new NameComponent[1];
            ncArray[0] = new NameComponent("Registration","");
            regObject = rowc.RegistrationHelper.narrow(nc.resolve(ncArray));

            // Activate CamManager
            activateCamManager(orb);
        }
        catch (org.omg.CORBA.TRANSIENT e)
        {
            e.printStackTrace();
            JOptionPane.showMessageDialog(null,
                "Unable to locate Naming Service.\nMake sure Naming Service is running.",
                "Cam Processor Error", JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            JOptionPane.showMessageDialog(
                null,"Error finding Registration.\nMake sure the CORBA server is running.",
                "Cam Processor Error", JOptionPane.ERROR_MESSAGE);
        }

        final CamProcessor app = new CamProcessor();
        app.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                quitting = true;  // Setting this will cause transmitter thread to quit

                try
                {
                    camManager.stopRec();

                    // Deregister with server since we'll no longer be available and unbind from
                    // Naming Service or CamManager for this location will remain

                    regObject.removeWebCam(location);
                    ncRef.unbind(camManagerNC);

                    // Stop and close the live video window
                    app.videoPanel.stop();
                }
                catch (Exception ex)
                {
                    ex.printStackTrace();
                }

                ((com.ooc.CORBA.ORB)orb).destroy();

                System.exit(0);
            }
        });
    }
```

```
    // Internal class
    class ViewingThread extends Thread
    {
        public void run()
        {
            // Create a DataSource that will be cloned by the VideoTransmitter
            try
            {
                DataSource ds;
                MediaLocator ml;

                // VFW is for Windows.  Use "sunvideoplus://0/1/JPEG" for Solaris
                if (fileSpecified == null)
                    ml = new MediaLocator("vfw://0");
                else
                    ml = new MediaLocator("file://" + fileSpecified);

                ds = Manager.createDataSource(ml);
                dataSource = Manager.createCloneableDataSource(ds);

                // Only open videoPanel once the vt's dataSource is set...
                if (!videoPanel.open(dataSource))
                    showError("Unable to play the data source.");
                else
                    videoScrollPane.revalidate();   // Makes scroll pane show its contents
            }
            catch (Exception e)
            {
                e.printStackTrace();
                if (fileSpecified == null)
                    showError("Unable to connect to capture device.\nA compatible web cam must "+
                        "be installed for this application to stream video.");
                else
                    showError("Unable to open file " + fileSpecified + " to display.");
            }

        }
    }
}  // End CamProcessor



// Must be external class because it is instantiated using a public static method.

class TransmitterThread extends Thread
{
    private String hostName;

    TransmitterThread(String hostName)
    {
        this.hostName = hostName;
    }

    public void run()
    {
```

```
        CamProcessor.vt = new VideoTransmitter(
            ((SourceCloneable)CamProcessor.dataSource).createClone(), hostName, "5050");

        String result = CamProcessor.vt.start();
        if (result != null)
        {
            System.err.println("Failed to start VideoTransmitter: " + result);
            return;
        }
        else
            System.out.println("Started VideoTransmitter.");

        CamProcessor.lblInfo.setText("Transmitting to host " + hostName + " on port 5050.");

        Thread thisThread = Thread.currentThread();
        while (CamProcessor.transThread == thisThread && !CamProcessor.quitting)
        {
            try
            {
                thisThread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }

       CamProcessor.vt.stop();
       CamProcessor.lblInfo.setText("");
    }
}
```

```
/*  CamManagerImpl.java
 *
 *  Object is created by a CamProcessor and used by a CamController to control
 *  a web cam remotely.
 *
 *  Copyright (C) 2002 by Frank McCown
 *  University of Arkansas at Little Rock
 *  July 2002
 */

import java.io.*;
import java.util.*;
import rowc.*;

public class CamManagerImpl extends CamManagerPOA
{
    protected Hashtable callbacks;
    protected CamVideoCallback callback;
    private String camLocation;  // location of cam manager
    private RTPExport export;
    private String serverHostName = null;

    private final String TEMP_VIDEO_FILENAME = "temp.mov";

    private boolean transmittingVideo = false;
```

```
private boolean recordingVideo = false;


public CamManagerImpl(String location)
{
    camLocation = location;

    export = new RTPExport();
    callbacks = new Hashtable();
}

// Need server's host name to connect with socket
public void setServerName(String hostname)
{
    serverHostName = hostname;
}


/**************************************************************************
 *
 *  Remote method implementations
 *
 **************************************************************************/

// Start transmitting live video stream to CamController at given ipAddr.
// The callback is used for notifiying CamController when the transmission
// has begun.  Return false if already bust transmitting, true otherwise.
public synchronized boolean startTransmission(String hostName, CamVideoCallback callback)
{
    if (transmittingVideo)
        return false;

    transmittingVideo = true;

    // The callback gets called when video is finished
    this.callback = callback;

    // Start sending video to ipAddr and notify receiver of incoming RTP transmission
    String url = "rtp://" + hostName + ":5050/video";

    System.out.println("sending to "+url);

    CamProcessor.transmitVideo(hostName);

    callback.videoAvailable(url);

    return true;
}


// Stop transmitting live video stream to CamController.
public void stopTransmission()
{
    transmittingVideo = false;

    CamProcessor.stopTransmitVideo();
```

```
}

// Start recording to file.  Return false if already busy recording to file,
// true otherwise.
public boolean startRec(int maxRecTime, CamVideoCallback callback)
{
    System.out.println("startRec called");

    if (recordingVideo)
        return false;

    recordingVideo = true;

    // The callback gets called when video is finished
    this.callback = callback;

    // Start in new thread so we can return from this function immediately.  This allows
    // stopRec to be called.
    ExportThread et = new ExportThread(maxRecTime);
    et.start();

    return true;
}

// Send video file from CamProcessor to MediaServer using sockets.
private void transmitFile()
{
    BufferedOutputStream fos = null;
    java.net.Socket socket = null;

    try
    {
        // open a socket connection.  Need server host name and socket
        socket = new java.net.Socket(serverHostName, FileServer.FILE_SERVER_PORT);

        // open I/O streams for objects
        fos = new BufferedOutputStream(socket.getOutputStream());

        File transferFile = new File(TEMP_VIDEO_FILENAME);
        BufferedInputStream fin = new BufferedInputStream(new FileInputStream(transferFile));

         // First write out location so Server can name file.
         String fileName = camLocation + "\n";
         byte byteArray[] = new byte[fileName.length()];
         byteArray = fileName.getBytes();

         fos.write(byteArray);

         int c = fin.read();
         while (c != -1)
         {
             fos.write(c);
             c = fin.read();
         }

         fin.close();
```

```
            fos.close();
        }
        catch(Exception e)
       {
            System.out.println("Error transmitting file to Server.");
            e.printStackTrace();
       }
}

// Stop recording video stream.
public void stopRec()
{
    export.stopExport();
    recordingVideo = false;
}


// Thread to export video to file, transmit to MediaServer, and notify CamController
class ExportThread extends Thread
{
    private int maxRecTime;

    ExportThread(int maxRecTime)
    {
        this.maxRecTime = maxRecTime;
    }

    public void run()
    {
        // Generate output media locators

        String currentDir = System.getProperty("user.dir");
        String fileName = currentDir + "/" + TEMP_VIDEO_FILENAME;

        javax.media.MediaLocator oml = new
            javax.media.MediaLocator("file:/" + fileName);

        CamProcessor.lblInfo.setText("Recording to file... ");

        // Must create clone of data source so video continues to play in CamProcessor.
        if (!export.startExport(
            ((javax.media.protocol.SourceCloneable)CamProcessor.dataSource).createClone(),
            //CamProcessor.dataSource,
            oml, maxRecTime))
        {
            System.err.println("RTPExporting failed");
            VideoClip video = new VideoClip("","",0,0,0);
            callback.newRecording(video);
        }
        else
        {
            // Notify client of new media

            VideoClip video = new VideoClip();
            video.location = camLocation;
            video.timeStamp = System.currentTimeMillis();
```

```
                // Name is taken from timestamp.  Replace spaces and : with underscores
                video.name = new java.util.Date(video.timeStamp).toString();
                video.name = video.name.replace(' ', '_');
                video.name = video.name.replace(':', '_');

                File f = new File(TEMP_VIDEO_FILENAME);
                video.size = (int)f.length();

                video.length = export.actualRecTime; //maxRecTime;

                // Now that video is produced, send it to Server for storing using sockets

                CamProcessor.lblInfo.setText("Transmitting to server... ");
                transmitFile();

                CamProcessor.lblInfo.setText("");

                // Notify CamController of new video
                callback.newRecording(video);

                recordingVideo = false;
            }
        }
    }  // end ExportThread
}
```

References

[ACE02] ACE ORB, The, May 2002, <http://www.cs.wustl.edu/~schmidt/TAO.html>.

[App02] Apple Computer, QuickTime, June 2002, <http://developer.apple.com/quicktime>.

[Bin02] Binet, G., JMF interest mailing list, May 2002, June 2002, <http://swjscmail1.java.sun.com/cgi-bin/wa?A2=ind0205&L=jmf-interest&P=R22937>.

[Bla02] Blackdown, “Java Media Framework for Linux,” May 2001, June 2002, <http://www.blackdown.org/java-linux/jdk1.2-status/jmf-status.html>.

[DD02] Deitel, H. M. and P. J. Deitel, Java How To Program, 4th ed., Prentice Hall, 2002, pp. 660-661.

[EKP00] Eliassen, F., T. Kristensen, T. Plagemann, H. Rafaelsen, “MULTE-ORB: Adaptive QoS Aware Binding,” 2000, June 2002, <http://unik.no/~tomkri/papers/eliassenrm00.ps>.

[Gem95] Gemmell, D. J., “Multimedia Storage Servers: A Tutorial,” IEE Computer Magazine, vol. 8, no. 5, May 1995.

[GHJ95] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, pp. 4-6.

[GS98] Gokhale, A., and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," Transactions on Computing, Vol. 47, No. 4, 1998.

[HSK97] Hong, J., Y. Shin, M. Kim, J. Kim, and Y. Suh, "Design and Implementation of a Distributed Multimedia Collaborative Environment," 1997, June 2002, <http://dpnm.postech.ac.kr/maestro>.

[ION02] IONA Technologies, ORBacus for C++ and Java, June 2002, <http://www.iona.com/products/orbacus_home.htm>.

[JMF02-1] JMF interest mailing list, June 2002, <http://swjscmail1.java.sun.com/cgi-bin/wa?A2=ind0206&L=jmf-interest&D=0&P=2733>.

[JMF02-2] JMF interest mailing list, June 2002, <http://swjscmail1.java.sun.com/cgi-bin/wa?A2=ind0107&L=jmf-interest&P=R34476&m=12699>.

[LC99] Li, S., and W. Chen, "A Java-centric Distributed Object-based Paradigm for Surveillance Services and Visual Message Exchange," Journal of Visual Languages and Computing, No. 10, 1999.

[LH00] Lai, B., and H.E. Hanrahan, "The Design of a TINA based Stream Management/Binding Framework," SATCAM 2000 Proceedings, July 2000.

[Log02] Logitech QuickCam, June 2002, <http://www.quickcam.com/html/index2.html>.

[Mic02-1] Microsoft, Windows Platform SDK: Video For Windows, June 2002, <http://msdn.microsoft.com>.

[Mic02-2] Microsoft, Streaming Methods: Web Server vs. Streaming Media Server, June 2002, <http://www.microsoft.com/Windows/windowsmedia/compare/webservvstreamserv.asp>.

[Mou96] Mourad, A., "Doubly-striped Disk Mirroring: Reliable Storage for Video Servers," Multimedia, Tools and Applications, Vol. 2, May 1996, pp. 273-279.

[MSS02] Mungee, S., N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," Jan 1999, May 2002, <http://www.cs.wustl.edu/~schmidt/research.html>.

[NOO02] NOOCEO, ezlinX, June 2002, <http://www.nooceo.com>.

[OMG00-1] OMG, "CORBAservices: Notification Service Specification," OMG Document formal/00-06-27, Object Management Group, Framingham, MA, June 2000.

[OMG00-2] OMG, "CORBAservices: A/V Streams," OMG Document formal/00-01-03, Object Management Group, Framingham, MA, Jan 2000.

[OMG01] OMG, “CORBAservices: Event Service Specification,” OMG Document formal/01-03-01, Object Management Group, Framingham, MA, Jan 2001.

[OMG02] Object Management Group, http://www.omg.org/news/about, June 2002.

[PHS96] Pyarali, I., T. Harrison, and D. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging,” USENIX, Vol. 9, No.3, 1996.

[Sca01] Scallan, T., A CORBA Primer, Segue Software, Feb. 2001.

[SCF96] Schulzrinne, H., S. Casner, R. Frederick, and V. Jacobson, “RFC 1889: RTP: A Transport Protocol for Real-Time Applications,” January 1996, June 2002, <http://www.ietf.org/rfc/rfc1889.txt?number=1889>.

[SCW99] Stiller, B., C. Class, M. Waldvogel, G. Caronni, D. Bauer, B. Plattner, “A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience,” IEEE JSAC: Special Issue on Middleware, vol. 17, no. 9, September 1999, pp. 1580-1598.

[Sun02-1] Sun Microsystems, “Supported Media Formats and Capture Devices”, June 2002, <http://java.sun.com/products/java-media/jmf/2.1.1/formats.html>.

[Sun02-2] Sun Microsystems, Java Remote Method Invocation, June 2002, <http://java.sun.com/products/jdk/rmi/index.html>.

[Sun02-3] Sun Microsystems, Java Web Start, June 2002, <http://java.sun.com/products/javawebstart/index.html>.

[Sun02-4] Sun Microsystems, Java Media Framework, June 2002, <http://java.sun.com/products/java-media/jmf/index.html>.

[Sun02-5] Sun Microsystems, “Java Remote Method Invocation – Distributed Computing for Java,” White Paper, June 2002, <http://java.sun.com/marketing/collateral/javarmi.html>.

[Sun99] Sun Microsystems, JMF 2.0 API Guide, “Working with Real-Time Media Streams,” November 1999, June 2002, <http://java.sun.com/products/java-media/jmf/2.1.1/guide/RTPRealTime.html>.

[TS02] Tanenbaum, A. and M. Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2002, pp. 504-505.

[Wil02] J. R. Wilcox, Videoconferencing and Interactive Media: the Whole Picture, Telecom Books, 2000, p. 169.

[WHZ01] Wu, D., Y. T. Hou, W. Zhu, Y. Zhang, J. Peha, “Streaming Video over the Internet: Approaches and Directions,” IEEE Transactions on Circuits and Systems for Video Technology, Vol. 11, No. 3, March. 2001, pp. 282-300.

[ZG01] Zhu, W. and N. D. Georganas, “JQoS: Design and Implementation of a QoS-based Internet Videoconferencing System using the Java Media Framework (JMF)”, 2001, June 2002, <http://www.mcrlab.uottawa.ca/papers/CCECE2001.pdf>.