THE RTMM TOOLBOX FOR DMM APPLICATIONS

Robin Sharp and Edward Todirica

Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Kgs. Lyngby, Denmark.

ABSTRACT

This paper describes an approach to implementing distributed multimedia applications based on the use of a software toolbox. The tools in the box allow the designer to specify which components are to be used, how they are logically connected and what properties the streams of data to be passed between the components are to have. Examples are given of system components for handling audio, video and networking, and the performance offered by this approach is discussed.

1. INTRODUCTION

At the Technical University of Denmark, work is in progress to develop a Virtual Seminar Room (VSR) as an example of an interactive distributed multimedia (DMM) system. It is well known that implementation of systems of this type involves a number of technical challenges, as it is necessary to ensure that data representing video pictures, still images, sound and other information are passed between the sites taking part in the seminar, and are presented in real time to the users at these sites, to give them the illusion that they are taking part in a discussion in the same room. To do this, it is necessary to have an efficient system for capturing, distributing and replaying high quality video, audio and other data in real time, in order to meet the desired synchronisation requirements [1].

The approach to system construction taken in the RTMM project is to provide the DMM application implementor with a software toolbox based on a number of simple concepts. The toolbox offers facilities for connecting active entities, known as *system components*, via logical channels through which *streams* of data with various quality of service requirements can be passed. Typically, a system component is composed of a hardware device and adaptor card with its driver, but pure software components, such as stream transformers, also occur. The facilities are offered at the interface to a conceptual *Stream Layer*, which is the basis on which all applications are built. For portability, the Stream Layer software is implemented as a GNU C++ library, and the system runs on a standard Linux/PC platform.

2. THE STREAM LAYER ABSTRACTION

The Stream Layer offers its users facilities for defining system components, for specifying how they are logically connected and for specifying which types of streams of data pass through these connections. Figure 2 shows a simple example of a system consisting of two physically separated sites, each containing two system components: a video system component and a network system component. A *stream* is a logically related sequence of data units which pass between two system components, starting at the *source* of the stream and finishing at its *destination*.

A *socket*, through which the stream passes, is used to identify the particular starting point of a stream in the source system component and the particular ending point in the destination system component. RTMM sockets permit unidirectional flow of data. The socket in the source from which the stream starts is denoted an *OutSocket* and the socket in the destination at which the stream ends an *InSocket*. For illustrative purposes, these are designated by graphical symbols as shown in Figure 1.

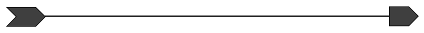

Figure 1: A connection from an OutSocket (left) to an InSocket (right)

To pass a stream between system components, a *connection* must be set up between these components. When initially created, a connection connects two sockets: an *OutSocket* and an *InSocket*, which may be associated with the same or different system components. By setting up a connection to one or more further *InSockets*, an *OutSocket* may become the source for a stream with multiple destinations. However the same *InSocket* cannot be the destination for several streams.

A connection passing between two sockets offers a certain *Quality of Service (QoS)* to the stream which it carries. The parameters of the QoS include the bandwidth, delay, jitter and other properties of the connection, and are expected to be compatible with the QoS required for the stream.

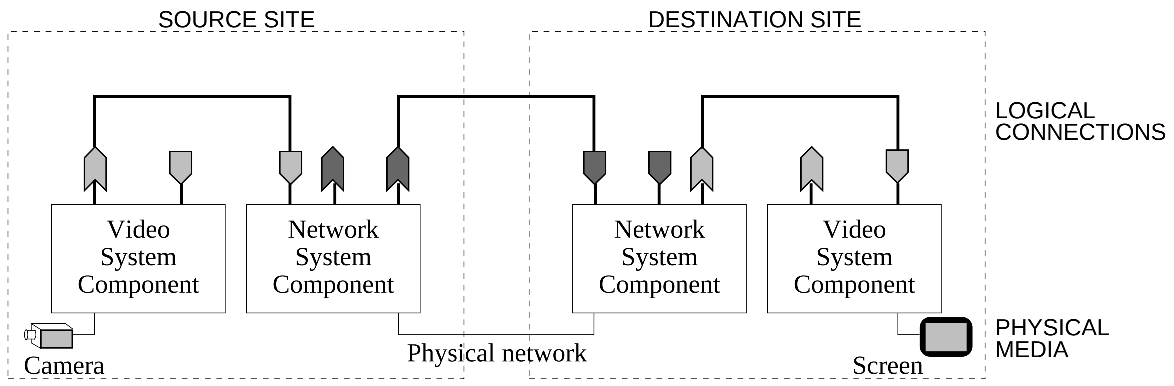

Figure 2: A system with two sites and four system components.

A more complete description of the Stream Layer API is given in [2].

3. SYSTEM COMPONENTS

Conceptually, each system component can have one or more InSockets through which incoming streams can be received by the component, and one or more OutSockets through which outgoing streams can be sent out of the component. An incoming stream can be *consumed* by the component, in which case the data in the stream are typically directed to some ouput device associated with the component. Similarly, a component may be used to *generate* an outgoing stream, whose data typically originate from some input device. Finally, a component may pass an incoming stream on as an outgoing stream, possibly after *transforming* it into a stream with new properties. The Stream Layer API offers functions for associating generator (source) functions, consumer (sink) functions and transformer functions with the component, and for setting up internal connections within the component in order to pass incoming streams at InSockets to OutSockets via which the streams can leave the component.

A typical RTMM DMM system is composed of several different types of system component. Most of these are directly associated with particular hardware adapter cards based on specialised auxiliary processors that are exploited to reduce the load on the CPU. Examples used within the implementation of the VSR set up at DTU are:

Audio A: Based on a standard stereo sound card for encoding and decoding audio signals.

Audio B: Based on the Spinner DSP board which is used for executing more advanced signal processing algorithms for source location, beam forming and echo cancellation, using a 4-microphone array as input source and a two channel audio setup for output [3].

Video A: Based on the Matrox Marvel G400 graphics card for MJPEG coding and a standard software library (IJG JPEG) for MJPEG decoding of video in SIF/CIF format.

Video B: Based on the Equator MAP-CA video DSP platform for more advanced video processing, including MJPEG en/decoding in full PAL resolution and MPEG en/decoding.

UDP Network: Based on a standard network card and UDP-/IP protocol stack.

As can be seen, some of the components are implemented in several versions, which can be selected according to the desired quality of the relevant stream and/or the target price of the complete system.

Figure 3 shows an example of the code needed to set up a small application involving a video component, an audio component and a network component on each site. The corresponding system structure is shown in Figure 4.

4. PERFORMANCE

A series of timing measurements made on the system are summarised in Table 1. These timings were observed over a two-way link between DTU and Aarhus in Denmark, a distance of about 400 km., using UDP/IP over the Danish Research Network. The computers at both ends are based on a 933 MHz Pentium III CPU, runnning RedHat Linux version 6.2 with the Linux 2.3.99 kernel. In all the measurements,

	Min.	Max.	Avg.	Std.dev.
Stream layer	0.03	0.52	0.08	0.03
Video processing	26.4	33.2	27.8	0.52
Network transmission	10.2	15.0	11.2	0.48
End-to-end video	37.3	44.8	39.2	0.72

Table 1: Processing times in the system (milliseconds)

```
startStreamLayer();
//Declare components
VideoComp C1;            //Video component
NetComp   C3;            //Network component

//Create sockets for the components.
//Socket id: first  number - component no.
//           second number - stream no.
//Initial "s": local socket, "n": network socket
OutSocket* s11 = C1.createOutSocket("s11");
InSocket*  s12 = C1.createInSocket ("s12");
InSocket*  s31 = C3.createInSocket ("s31");
OutSocket* s32 = C3.createOutSocket("s32");
OutNetSocket* n31 =
    C3.createListenSocket("Stream1","n31");
//Set up socket to receive on agreed port
InNetSocket*  n32 = C3.createReceiver("45678");

//Create connections between components.
//Connection id: first  number - source comp.
//               second number - destn. comp.
Connection *c13,*c31;
try{ c13= new Connection("Stream1",s31,s11);
     c31= new Connection("Stream2",s12,s32);
} catch (Connection_error e){e.showError();};
//Create internal connections in C3.
C3.connectInToOut("",0,0,s31,n31);
C3.connectInToOut("",0,0,n32,s32);
//Create network connection to remote site
n31->connect("foo.bar.com 45678");

//Open connections to allow stream to flow
c13->openConnState();
c31->openConnState();

//Associate source + sink functions with sockets
try{ C1.source(1,"Stream1",0,s11);
     C1.sink(1,"Stream2",0,s12);
}catch (Component_error e){e.showError();}

//Activate the generated stream
try{C1.activateStream("Stream1");
}catch (Component_error e){e.showError();}
  ...
  ...
endStreamLayer();
```

Figure 3: Code for setting up a small application using the toolbox

Only the code related to the video streams is shown; the code for the audio streams is similar.

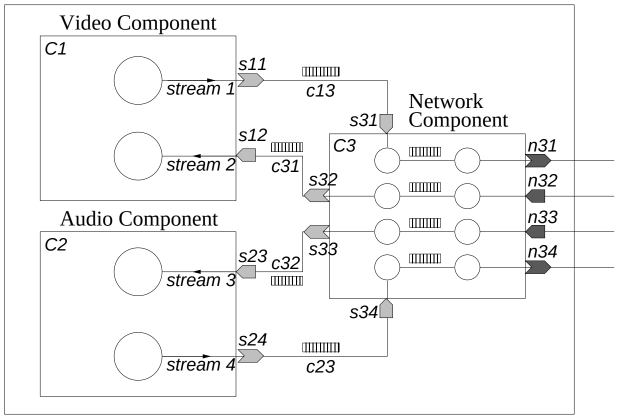

Figure 4: The software architecture of a simple site

Circles symbolise threads and striped rectangular boxes buffers (queues). The component, connection and socket identifiers correspond to the code example given in Figure 3.

the Video A video system component, operating in 352x288 pixel format at 25 frames/s, and the Audio A audio system component were used. The minimum, maximum and average bandwidth requirements for video were 3.15, 3.50 and 3.30 Mbit/s, with a standard deviation of 0.04 Mbit/s. For audio, the bandwidth was a constant 670 kbit/s for uncompressed mono CD quality. All measurements refer to a period of about 1 minute (1500 frames). For network transmission, one IP packet was used to contain each JPEG encoded video frame (about 16.5 kbytes) or a mono audio packet of about 1 kbyte.

The times referred to in the table are measured as follows:

- Stream layer: On source site, the time from the start of transmission of a frame from the source video system component until the end of receipt by the source network system component. This is a measure of the delays introduced by the Stream Layer software.
- Video processing: In the destination system component, the time from the instant when the frame is completely received by the component until the decoded frame has been stored in the frame buffer.
- Network transmission: The time from start of transmission of a frame from the source network system component until the entire frame has been received by the destination network component. (This is measured as half the time for sending the frame to the remote site and back again, where the remote site acts as a reflector.)
- End-to-end video: Measured as the time from the instant when the frame is completely captured until the presentation function returns, where the video frames are sent to the remote site and back again, with the remote system functioning as a reflector. The network transmission time is here halved, so the value in the table is a measure of the true end-to-end delay for video between two sites.

Not unexpectedly, these times are completely dominated by the video processing and network transmission times. The overhead introduced by the Stream Layer for transfer of data between system components on the same site is very small.

5. DISCUSSION

The performance of the system depends strongly on the implementation of the Stream Layer software, and in particular on the way in which the various activities are partitioned on threads. The current implementation associates a thread with each socket. On each site, data from a particular stream are passed from the source system component through one or more connections to a network OutSocket or from a network InSocket through one or more connections to the sink system component. For the system set up by the code of Figure 3, this is illustrated in Figure 4.

Each connection contains a queue with a limited size (default size 10 elements). The thread corresponding to the OutSocket puts data units on the queue, and the thread corresponding to the InSocket gets data units from the queue. The queue operations are blocking, so the OutSocket thread is blocked when the queue is full and the InSocket thread is blocked when the queue is empty. In this way, even if there are many threads in the VSR application, only a few of them are active at any given time; most of them are blocked waiting on empty queues. This means that the scheduler has a relatively simple problem of handling only a few ready to run threads. We can say that in the Stream Layer the activity follows the data units: threads become active only if they have a data unit to transfer. Thus the threads corresponding to the Stream Layer have a very short execution time, and the total overhead due to the Stream Layer is very small, most of the time being used by the threads corresponding to sources and sinks.

Neither the Linux OS nor the UDP/IP network offer QoS guarantees, so the resources available for the VSR application are strongly influenced by the other activities in the system. In such a best-effort system, it is important not to have many competing tasks running together with the VSR application. In particular, the software video decoding technique is very CPU intensive and competition with other high-CPU activities can cause loss of frames due to failure to meet decoding deadlines. However, it is, as we see, in practice possible to achieve good results, with low ($\sim$40 ms.) end-to-end latency, full 25 frames/s frame rate and low ($< 1 \times 10^{-3}$) rate of frame loss, in a system with a moderate load – for example, a window manager and a few other non-CPU intensive applications. This is quite adequate for the kind of activity which needs to be supported while the VSR is in operation.

6. CONCLUSIONS

A software toolbox implemented on the basis of a standard operating system has been shown to offer designers of multimedia applications a convenient way of structuring their designs. The run-time penalty introduced by using this approach is very small compared to the time required for decoding and network transmission in the system. It offers a lot of flexibility by allowing easy integration of different data sources and sinks. As the example presented in this paper illustrates, the Stream Layer is also easy to use. It reduces the development time for multimedia applications by letting developers worry about how they should produce and present their streams and not about how to transfer the streams efficiently.

Creation of this toolbox has not involved modification of the Linux operating system kernel. This makes it simpler to deploy the toolbox, as it avoids kernel patches and problems caused by incompatibilities amongst various kernel versions. It also makes the toolbox portable. Although built for Linux, a version for Windows could easily be produced by using the Cygwin environment. The toolbox has been applied in the RTMM VSR project, where it is controlled via a GUI which enables the user to ”plug together” components, both locally and remotely. We intend to continue to experiment with this approach and are currently developing further system components, including a virtual whiteboard, a virtual slide projector and an ATM network system component, for use in the system in the VSR and other multimedia applications.

7. ACKNOWLEDGMENTS

The RTMM project has been partially supported by the Danish Research Councils and the Danish Center for IT Research as part of the Center for Multimedia in Denmark.

8. REFERENCES

[1] Ralf Steinmetz, “Human perception of jitter and media synchronization,” *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 1, pp. 61–72, Jan. 1996.

[2] Robin Sharp, Hans-Henrik Løvengreen, and Edward Todirica, “Streams and sockets in DTU-RTMM, version 1.4,” Tech. Rep., Department of Information Technology, DTU, Sept. 2000.

[3] S. Forchhammer, A. Fosgerau, P.S.K. Hansen, R. Sharp, E. Todirica, and A. Zsigri, “Video conferencing for a virtual seminar room,” in *Proc. 4th International Conference on Digital Signal Processing and its Applications, Moscow*, Feb. 2002.