Lightweight Scalable Tool Sharing for the Internet

Agustín José González
Electronic Engineering Department
Universidad Técnica Federico Santa María
Valparaíso, Chile
agv@elo.utfsm.cl

Hussein Abdel-Wahab and J. Christian Wild
Computer Science Department
Old Dominion University
Norfolk, VA, USA.
(wahab,wild)@cs.odu.edu

Abstract

Increased desktop processing power and network bandwidth have made feasible distributed multimedia collaborative systems. Such systems are characterized by video, audio and data exchanges. While there has been much research and development of protocols and services for video and audio transmission, less work has been focused on data sharing particularly when the data is being generated by an application in real time and in turns controlled by one or more of the participants in the collaborative session. This paper describes a protocol and its implementation for sharing desktop applications in a distributed collaborative session. Key objectives of this protocol are scalability and resilience to dropped packages and to participants joining and leaving a session. We describe design decisions and give results demonstrating the effect two different compression algorithms and protocol parameters. Finally, Odust a tool sharing system built on the implementation of shared application views is described.

1. Introduction

The growth of the Internet and the increased performance of desktop computers have made feasible large-scale multimedia applications over communications networks. This work focuses on synchronous collaborative multimedia applications. Synchronous collaboration takes place when the participants involved in common tasks are seated simultaneously at their workplaces. This is the case of virtual-classroom systems for distance learning and multimedia conferencing systems. In this work we describe a distributed tool for sharing applications to be used in conjunction with video and audio in these scenarios.

Synchronous multimedia applications are based on three basic components: audio, video, and shared data. The transmission of audio and video information over digital networks has been studied for years and the field is relatively mature. However work on the transmission of data from interactive applications to a large group is still a relatively new area of investigation. The information shared in a collaborative session is often one of the main foci of the session. Rather than sending hard copies or faxing the material to remote participants, today's collaboration systems use the network to distribute this information on the fly. Many specialized systems have been developed for that purpose, such as co-browsers [3] [6], and sharing tool engines [1] [13]. In other cases, the collaboration application includes a module for data sharing such as in [10] [9] [11]. One approach for tool sharing enables one to share existent unmodified single-user applications. Examples of such systems are X Teleconferencing and Viewing (XTV) [1], Virtual Network Computing (VNC) [13], and Java Collaborative Environment (JCE) [2]. Our work follows a similar approach to that taken by VNC, which achieves tool sharing by distributing the desktop as an active image. This means an image which users can interact with in a similar way they do with their local desktop. Rather than sending one entire desktop as VNC does, we propose to send the images of the actives windows of applications running on any of the participants' screens. Unlike the other approaches our transmission protocol relies on IP multicasting to achieve scalability and a video-like scheme to overcome packet losses. In this work, we describe a protocol for sending dynamic images and Odust, a distributed sharing tool application based on this protocol. In Odust, users at the receiving sites can also request a floor and control the shared tool remotely. Our goal in designing this Odust was to provide a tool for interactive distance learning systems and large group conferences on the Internet where it should work in conjunction with distributed audio and video tools. In such scenarios, we used Java and a minumum of unavoidable architectures dependency to achieve portability. Also, the robustness of our distributed protocols provide graceful degradations in face of users or communications failures.

The rest of this paper is organized as follows. The next section describes the Dynamic Image Transmission Protocol giving results of experiments that were used in developing and tuning its implementation. Section 3 describes a tool sharing service which uses the dynamic image transmission protocol. This is followed by a section giving related work and a conclusion.

2. Dynamic image transmission protocol

The protocol for transmitting dynamic images presented here enables data sharing by disseminating images generated by shared interactive applications. From the communication point of view, the two main features of this protocol are resiliency and scalability. Resiliency refers to the ability of the protocol to overcome transmission losses and to accommodate the leaving and joining of participants. Scalability is achieved by eliminating the need for acknowledgements or any feedback from the receivers. The set of views of a running application can be thought of as a video stream of dynamic images and can utilize existing video compression technology. Our protocol tiles the image in square blocks, and then it encodes each block using a standard image coding to remove spatial redundancy. Only blocks that change between two image samples are encoded, thus some temporal redundancy is also removed. There are several advantages to dividing the image into tiles that relate to error recovery, late joins, and transmission efficiency. To remove temporal redundancy motion-compensated prediction [5] has been used in video encoding. Motion prediction is computationally intensive and has limited utility for the synthetic video streams generated by computer applications when the sample rate is low, therefore we have chosen to only use motion prediction with null motion vector. Thus blocks that remain unchanged from one sample to another are not resent. In contrast to video, these images tend to be of higher and variable resolution than traditional video images and present lower degrees of motion. In addition, dynamic images often contain information which requires high fidelity rapidly becoming illegible as image quality is reduced. We call this property *legibility*. This is especially true for images containing text. We estimate that a sampling rate of around 2 samples per seconds fulfills the requirements of most types of data sharing. Given a bounded CPU allocation for data sharing, bigger picture processing can only be achieved by reducing the processing cycle rate.

Image size changes are also transmitted by the protocol. We observe that window resizing usually preserves the upper left content of the view regardless the side or corner used for resizing. Therefore, receivers initialize the new version of the image with the upper left content of the previous instance of the image.

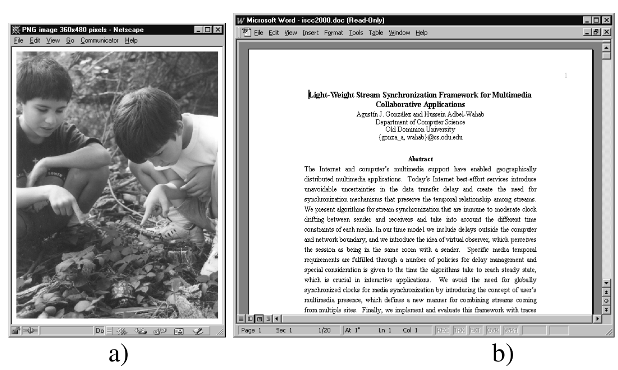
The use of unreliable transport protocol forces our us to take some precautions to overcome packet losses. We decided against selective retransmission of lost data because of its difficulties in getting feedback from an undetermined number of receivers [12]. Instead, we send new data or control the retransmission of the same data to eventually repair the original lost. We define the protocol data unit (PDU) such a way that each PDU can be processed out of order with respect to other PDUs. As a result, each PDU conveys at least a tile, its coordinates within the image, a tile-based sequence number, and timestamp. We schedule tiles retransmission after random times taken from (0, *update_tile*] sampling periods. The protocol accommodates late comers by sending a refresh for each PDU after a random time taken from the interval (*update_tile*, *refresh_time*]. This ensures a full image retransmission takes place at most every *refresh_tile* sampling periods.

Image creation is simple and signaled by the reception of the first PDU; however image removal is a little more involved since there is no guarantee that any explicit close image message will reach all the receivers. Close image messages are used in conjunction with a refresh image timeout to determine that the dynamic image was closed.

In the following sections we discuss the parameters of the protocols and their impacts on performance. First, we analyze the effect of two common compression standards for still image encoding that we tested for tile compression, and we discuss the tradeoffs in selecting the tile size. Then, we measure the protocol processing times and used them to estimate the sampling rate. Finally, we extend these ideas for the transmission of multiple possible overlapped images. To evaluate our protocol, we implemented it on Java 2 SDKv1.2.2 and employed Java Advanced Imaging 1.0.2 for tile compression [14].

2.1. Tile compression format study

We considered and compared Joint Photographic Experts Group (JPEG) [16] and Portable Network Graphics (PNG) [4] for tile coding. The criteria for selecting the compression technique were compression time, compression ratio, and legibility. These factors were evaluated as a function of the tile size around 32x32 pixels (the selection of tile size is covered in the next subsection).



a) b)

Fig. 1. Test images a) 388x566-pixel real-world image and b) 680x580-pixel Text image.

We compared two image types shown in Figure 1. While Figure *1a* is a real world picture, Figure *1b* shows a synthetic image containing text generated by a word processing program. These two types of images were picked as representative of the differences between natural and synthetic images. As shown in Table 1, for the natural world image, JPEG compression outperforms PNG by a factor of almost 10. For the synthetic image, PNG outperforms JPEG by a factor around 4.5. Due to the lossy nature of JPEG, a quality factor needs to be provided for compression. While quality values of around 50% are normally acceptable for JPEG pictures, higher values are required for legible text images. PNG, on the other hand, is lossless and offers good compression rates for text and line type of images, but it does not compress well the information of real-world scenes.

TABLE 1. PNG/JPEG comparison for two types of images.

Image	PNG size [Kbyte]	JPEG size [Kbyte]	PNG/JPEG Ratio
388x566 Real-wold	340	36 (Q=50%)	9.4
680x580 Text	21	94 (Q=75%)	0.22

We also compared JPEG and PNG for tile-size images. JPEG (at 50% quality) has an overhead of around 600 bytes while PNG overhead is near 100 bytes; however, JPEG image size grows with a lower slope compared with PNG's as image size grows. By overhead we mean the size of the compressed image as its size approaches zero.

As a result, there is lesser variation of compression ratio with JPEG, but it imposes a higher overhead in each tile. This defines the compression limit. As for these results, it appears that tiling decreases the performance; nonetheless, the counter argument is that smaller tiles enable more temporal redundancy removal.

2.2. Selecting tile size

The definition of the tile size has a crucial effect on performance. For PNG encoding, only 16x16-pixel tile size leads to a single network frame per packet for all tiles on Ethernet, and fragmentation is unavoidable for any other size on real-word images. For text-like images, on the other hand, PNG does a very good job in producing a single fragment even for 64x64-pixel tiles. In contrast, the average and maximum packet sizes do not vary much with the image content in JPEG. As a result, we selected JPEG compression and 40x40-pixel tile by being the biggest tile that does not lead to fragmentation on Ethernet. Fragmentation imposes a penalty not only on bandwidth but also in transmission processing time as we elaborate in the next section.

2.3. Protocol processing time

We measured the processing time using a prototype implementation. The two images shown in Fig. 1 were used is this comparison study. We experimented with the two compression techniques and different tile size. The main results are shown in TABLE 2. In our tests, compression introduced the dominant processing time. Notice these measures represent an upper bound for the two regions since we assumed a sequence of two identical images for comparison which forces the algorithm to touch every pixel, and two distinct images for compression which forces the protocol to compress every tile. Thus depending on the updates from one sample to another, the total processing time goes from 407 ms to 1.3 s for the real-world image.

TABLE 2. Sender's processing time using JPEG and 40x40 pixel tiles.

Step	Real-world Image (Fig. 1a)	Text Image (Fig. 1b)
Capture	100 ms (6%)	170 ms (6%)
Comparison	307 ms (19%)	556 ms (20%)
Compression	1132 ms (70%)	1889 ms (69%)
Transmission	78 ms (5%)	143 ms (5%)

This result shows that small updates can be sent at a rate of 2 Hz for this image size, and it takes up to around 2 seconds to send an entire new image. This upper bounds is directly proportional to the image size.

TABLE 3. Receiver's processing time using JPEG and 40x40-pixel tiles.

Step	Real-world Image (Fig. 1a)		Text Image (Fig. 1b)	
	WinNT	Solaris	WinNT	Solaris
Decom-pression	677 ms (86%)	483 ms (67%)	795 ms (79%)	826 ms (64%)
Updating Image	38 ms (5%)	46 ms (6%)	135 ms (13%)	85 ms (7%)
Displaying	75 ms (9%)	191 ms (27 %)	78 ms (8%)	364 ms (29%)

In contrast to the platform dependencies of the sender in this sharing tool application, the receiver part is fully coded in Java and run on WinNT or UNIX machines (Pentium II 266 MHz, 64 MB and Sun Untra 10 respectively). The results for both platforms are shown in TABLE 3. These are upper bounds because we expect a fraction of the tiles to remain unchanged. The dominant cost is decompression and the total time shows that the bottleneck is the sender side of the protocol.

3. Odust

Odust is a sharing tool engine based on dynamic image protocol for transmitting images of application windows.

We mention only WinNT even though we also mean Win95, and Win98. We have successfully tested receivers on WinNT and Solaris 2.6; however, recipients could be on any machine that runs Java and the Java Advanced Imaging package (JAI) [14].

Odust is a distributed cross-platform application that enables data sharing in synchronous multimedia collaboration. An owner of the shared application operates the real instance of it on the screen, while the other participants see and operate images, which are generated by Odust and are in many ways indistinguishable from the real application. Sharing is done with *process granularity* meaning that all the windows belonging to a process are shared atomically. A floor control service, described in [7], allows any receiver to request the control of the shared application by preempting it from the current holder. Due to the lightweight nature of the protocols for sending images and controlling the floor, any participant can leave the collaboration session at any moment. Likewise, anybody can join the session at any time. These two situations have virtually no effect on the other participants. Users joining the session late reach a synchronic view within a bounded time, which is a parameter in Odust. Multiple participants can simultaneously share their applications as long as only one tool is shared per site. Each shared tool is displayed in a separate window at the receiving user's desktop.

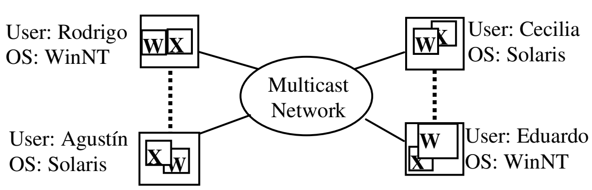


Fig. 2. Tool sharing scenario with Odust.

Fig. 2 shows one of the multiple scenarios where Odust can be used. In this scenario, four participants are sharing two computer applications. One application is an "xterm" (marked with an "X" in Fig. 2) which is running on user machine Eduardo. The second application is MS-WORD (marked with a "W" in the figure) which is running on user machine Rodrigo. Scalability is gained mainly due to the use of IP multicasting, which is a network requirement for Odust to work in more than 2 participant sessions. It also works over unicast network for 2-party sessions. The following figure illustrates the view that user Rodrigo of the Fig. 2 sees on the screen. The other views are similar, although each user can arrange the screen differently if desired (as suggested in Fig. 2).

Rodrigo shares an MS-Word application, as shown in Fig. 3. MS-Word runs outside Odust the same way it does any application on his machine. In addition, he receives the *xterm* being shared by Eduardo (owner label) but controlled by Agustín (leader label). Even though the *xterm* here is a UNIX application, it runs via an X Window-server on WinNT.

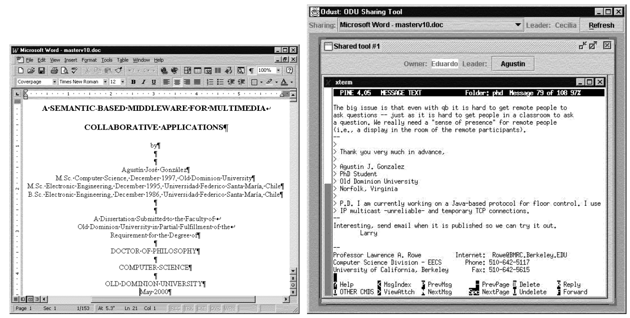


Fig. 3. The real MS-word application and Odust interface viewed by Rodrigo.

If other WinNT participants started sharing more applications, each participant would receive them in separate windows within Odust. This is the case of UNIX users in this scenario. They receive Rodrigo's MS-Word and Eduardo's xterm in different windows. Finally, floor control is done on per shared tool bases.

3.1. Overall architecture

Odust's architecture reflects the three main external features of it, application view dissemination, floor control, and remote tool interaction. A distributed object architecture implements the protocol for transmission of dynamic compound images. Another set of distributed object implements a lightweight floor control framework for centralized resources, which is described in [7]. Finally, two objects work in a client-service architecture to support the interaction with the shared application from remote sites. Odust depends on a single multicast group that is provided as command line argument. Now, in order to support multiple shared applications at a time, Odust multiplexes the multicast group in up to 256 channels. A distributed multiplexer-demultiplexer object dynamically manages channel allocation as new applications are shared. Each of the basic components of Odust, compound image transmission, floor control, and user's input events is made of two related objects. One centralized object resides on the machine sharing a tool and the others are replicated at every shared tool receiver. Fig. 4 focuses on the internal architecture of one sender and one receiver. The demultiplexer listens for messages coming on any channel. Multiplexer (Mx in Fig. 4) and demultiplexer (Dx in Fig. 4) are actually two Java interfaces implemented by the same class. Thus, each multiplexer can keep track of the channel in use and can randomly allocate a new unused channel when the local sender requests one to start transmitting a new shared tool to the session. As soon as its counterparts receive an

Application data Unit (ADU) from an unallocated channel, each receiver creates new *application receiver* object to process subsequent ADUs.

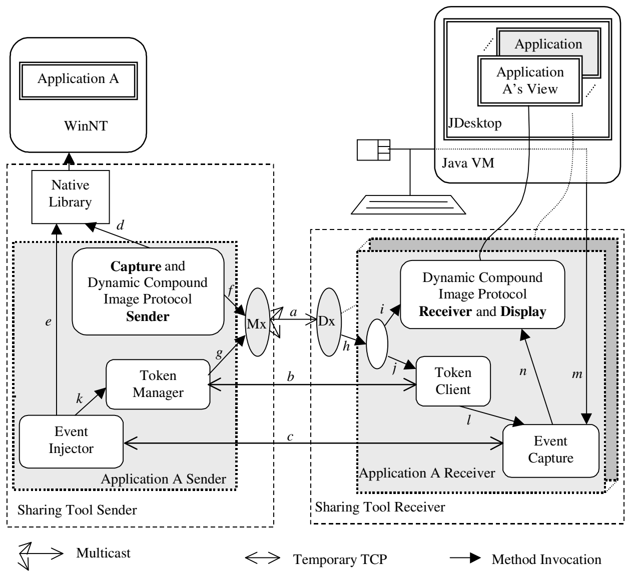


Fig. 4. Odust sender/receiver overall architecture.

Senders blindly transmit ADUs with no feedback from recipients. Both the image transmission and Token Manager objects share the same multicast channel. While the former transmits image protocol related messages, the latter periodically sends a heartbeat with the floor status (mainly floor holder), local host names, and Token Manager service port, so that clients can dynamically connect to the Token Manager (link *b* in Fig. 4). The native library implements three functions required by the Java capture object.

At the receiving site, the demultiplexer dispatches the ADUs to the corresponding *application receiver* according to the setting it saves when the *application receiver* is created upon receiving the first ADU (method call *h* in Fig. 4). Then, the *application receiver* dispatches the message to either the compound image receiver (method call *i*) or the Token Client (method call *j*).

Upon user's floor request, the Token Client connects to the manager and obtains the service access point of the Event Injector in the Grant message (*b* connection in Fig. 4). The Token Client forwards this information to the Event Capture object (method call *l*). Finally, connection *c* is established and the mouse and keyboard events of the new floor holder are sent to the application sender.

Connection *b* and *c* are only kept while the corresponding receiver holds the floor. The Event Capture object listens for input events within the application widget at receiving sites (method call *m*). When an input event is fired by the Java virtual machine, Event Capture forwards the event to its peer Event Injector as long as the event took place within one of the shared application images in the widget.

The Native Library is the only non-Java code. It implements 5 native methods that need to be ported to other platforms in order to share applications running on them; however, receiver's code has the same portability as Java code.

Even though the traffic due to the floor holder only affects two machines per floor in the session, we use mouse event filtering to reduce the number of events fired by mouse moves. Mouse movements are only sent to the application if they are far part in position or time. Two parameters govern the granularity of the filter.

4. Related work

Important sharing tool applications like XTV [1], JCE [2] and VNC [13] use TCP as transport protocol. In contrast, our protocol works on top of unreliable multicast transport layer. This makes a crucial difference that lets our protocol be considerably more scalable than the other proposals.

The idea of sharing data by sharing images has been explored in the VNC project [13] at Cambridge University. VNC's unit of transmission is, like our protocol, a rectangle of pixels. It uses raw-encoding or copy-rectangle encoding. In the first one, the pixel data for a rectangle is simply sent in left-to-right scan-line order. In contrast, we use still image compression for tiles. VNC avoids compression time but demands more transmission bandwidth than our protocol. Copy-rectangle encoding allows receivers to copy rectangles of data that are already locally accessible and can be used for motion prediction on synthetic images (e.g. scrolling). We decided against this type of primitive because of the high processing cost in determining tile motion. While VNC shares an entire common and unique desktop, we propose an application granularity for sharing which allows a participant to show only a selected application and keep the rest private. Finally, unlike VNC that shares only applications intalled on a central server, Odust allows any user to share her local platform dependent application.

Video Conferencing tool has also been used for data sharing by transmitting dynamic images as video frames. Its main advantage is the access to highly refined and tuned libraries for video streaming that reach higher frame rate than image processing. Another experience in sending data contents through video streams is found in *vic* version 2.8 [15] from University College London (UCL). One of the featured added at UCL allows the sender to select a region of the screen for frame capture as opposed to video frames. The video approach suffers from a number of shortcomings. First of all, video compression limits the video dimensions to a few sizes. This restricts its use when the information to be shared does not fit a predefined video size on the screen. On the other hand, the use of converters for sending the entire display view forces the sender to make the complete screen public. In addition, it inevitably reduces the resolution to, for example, 352x288 pixels in CIF (**C**ommon **I**ntermediate **F**orm) size video. Our protocol for transmitting dynamic images overcomes these shortcomings.

5. Conclusions and future work

Along with audio and video, data sharing is a crucial component in multimedia collaboration. In order to achieve data sharing, we developed a protocol for image transmission and used it to implement Odust, a sharing tool application. This resilient and scalable protocol compresses a sequence of image samples by removing temporal and spatial redundancy. Tiling and changes detection achieve the former, and a standard image compression technique accomplishes spatial redundancy removal. Protocol data unit losses are overcome by randomly re-transmitting tiles. This technique also provides support for latecomers.

This sharing tool application disseminates images of the shared application and accepts remote user input events as if they were coming from the local tool owner. It was tested on Win85, Win98, WinNT, and Solaris operating systems.

This work can be extended in two independent paths. One aims to reduce both processing time and bandwidth consumption of the protocol. The other approach is to adapt current video compression techniques to fulfill the requirements of data sharing. We are also considering H.263+ [5] video compression standard, which supports custom picture size. This feature removes one of the major drawbacks we have pointed out of video encoding and enables it for sharing images. Finally, we are porting the sampling library to other platforms to not only receive but also transmit from other platforms.

Acknowledgement

The authors wish to express their thanks for the financial support provided by the Chilean Research Fund (CONICYT), project number FONDECYT 1010100.

References

[1] H. Abdel-Wahab and M. Feit, "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration," in *IEEE Tricomm '91: Communication for Distributed Applications & Systems*, Chapel Hill, NC, USA, 1991. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 157-167, 1991.

[2] H. Abdel-Wahab, O. Kim, P. Kabore, and J.P. Favreau, "Java-based Multimedia Collaboration and Application Sharing Environment," in Proceedings of the Colloque Francophone sur I'Ingenierie des Protocoles (CFIP '99), Nancy, France, April 26-29, 1999.

[3] C.Bisdikian, S. Brady, Y.N. Doganata, D.A. Foulger, F. Marconcini, M. Mourad, H.L. Operowsky, G. Pacifici, and A.N. Tantawi, "Multimedia Digital Conferencing: A Web-enabled multimedia Teleconferencing system," *IBM Journal of Research and Development*, vol. 42, no.2, pp. 281-298, March 1998.

[4] T. Boutell, "PNG (Portable Network Graphics) Specification: Version 1.0," Request for Comments RFC 2083, January 1997.

[5] G. Côté, B. Erol, M. Gallant, and F. Kossentini, "H.263+: Video Coding al Low Bit Rate," IEEE Transactions on Circuits and Systems for Video Technology, vol. 8, no. 7, pp. 849-866, November 1998.

[6] J.Z. Davis, K. Maly, and M. Zubair, "A Coordinated Browsing System", Technical Report TR-97-29, Old Domimion University, Norfolk, VA, May 1997.

[7] A. González, "A Semantic-based Middleware for Multimedia Collaborative Applications," Old Dominion University, Norfolk, Virginia, Ph.D. dissertation, May 2000.

[8] W.T. Liou, J.J. Tan, and R.C.T. Lee, "Minimum Rectangular Partition Problem for Simple Rectilinear Polygons," IEEE Transaction on Computer-Aided Design, vol. 9 no. 7, pp. 720-733, 1990.

[9] K. Maly, H. Abdel-Wahab, C.M. Overstreet, C. Wild, A. Gupta, A. Youssef, E. Stoica, and E. Al-Shaer, "Distant Learning and Training over Intranets," IEEE Internet Computing, vol. 1, no.1, pp. 60-71, 1997.

[10] Microsoft's NetMeeting,http://www.microsoft.com/netmeeting.

[11] Microsoft's PowerPoint,http://www.microsoft.com/powerpoint.

[12] J. Nonnenmacher and E.W. Biersack, "Scalable Feedback for Large Groups," IEEE/ACM Transactions on Networking, vol. 7 no. 3, June 1999.

[13] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual Network Computing," IEEE Internet Computing, vol. 2, no.1, pp. 33-38, Jan/Feb 1998.

[14] Sun Microsystems, Java language, http://java.sun.com/products.

[15] Videoconferencing Tool, The Networked Multimedia Research Group at University College London, http://www-mice.cs.ucl.ac.uk/multimedia/software/vic/.

[16] G.K. Wallace, "The JPEG Still Picture Compression Standard," Communications of the ACM, vol. 34, no.4, pp. 30-44, April 1991.