Architectural Support for Online Multimedia Services

Hans Eberle[1] & Jürg Gutknecht[2]

[1]Sun Microsystems Laboratories
Hans.Eberle@eng.sun.com

[2]Swiss Federal Institute of Technology (ETH)
gutknecht@inf.ethz.ch

Abstract: Our vision is an environment that allows (ordinary) PC-clients to profit from a rich collection of multimedia programs such as selected TV channels, video on demand, teleported lectures etc. In this article, we present the design and implementation of a corresponding local infrastructure both from a hard- and software perspective. Innovation highlights are (a) a switch-based network with guaranteed transmission bandwidth for audio/video streams, built-in multicast support and globally accessible display frame buffers, (b) a system for the integrated display of remotely generated video streams at the client site and (c) a central server for the management of the available multimedia programs.

Keywords: client/server multimedia system, quality of service, networked peripherals, Oberon.

1. Introduction

Thanks to the power, versatility and low cost of today's personal computers multimedia has become an attractive and widespread field of application. The unique combination of high-quality memory-mapped display screens with customized software control gives general-purpose computers invaluable conceptual superiority against even most sophisticated television (TV) sets. In addition, new and more flexible kinds of data delivery services such as video on demand, video conferencing and process monitoring can be used much more profitably by computer-controlled clients.

A simple but useful multimedia environment consists of a set of audio/video stream sources (typically cameras, microphones, TV receivers or special-purpose processors), a program server/manager and a set of clients that may request programs from the server on demand. In this context, the only function of the client station is the display of (possibly multiple) video streams. It is therefore reasonable to aim at an architecture that saves both communication bandwidth and processor cycles by providing direct global access to the clients' frame buffers, under bypassing of their processors. Other desirable properties are the capability to multicast streams, to display moving images at high quality (no jerks or jitter) and to smoothly integrate remotely generated contents with the client's local display management.

Our hardware approach to these design goals is the switch-based system area network *Switcherland* [4]. In its most general form, Switcherland presents itself as a scalable distributed shared-memory network as depicted in Figure 1. Nodes appear in a variety of forms: processor/memory, frame buffer/display, audio/video digitizer, disk or other special purpose device. Interesting properties of the system area network are bandwidth guarantees and a built-in multicasting capability.

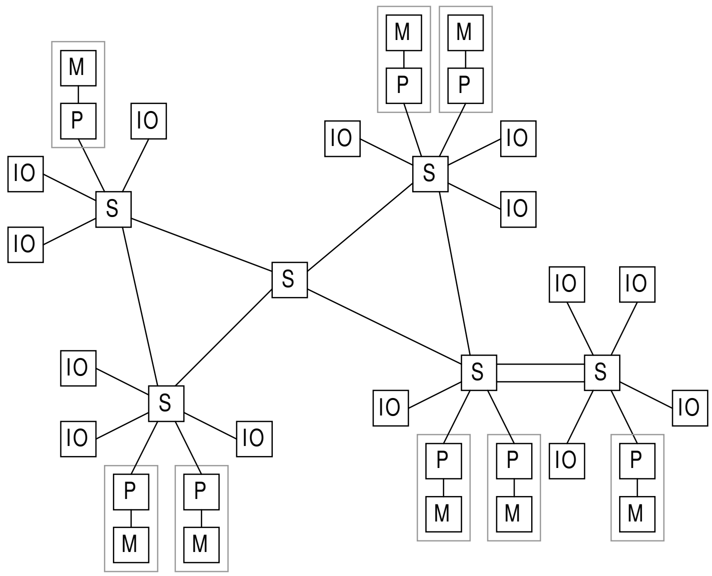

Figure 1: A distributed Switcherland system consisting of switches (S), processor/memory nodes (P/M) and IO nodes (IO).

At the client site we use off-the-shelf PCs connected to Switcherland via PCI adapters and proprietary display frame buffers. Clients run Oberon System 3 [5], an evolution of the original Oberon system [15][16]. System 3 runs on bare Intel hardware. It incorporates a framework for visual objects and an elaborate concept of a display space. Figure 2 shows a snapshot of a System 3 display screen. For the purpose of the multimedia project under discussion, the system had to be upgraded by a notion of remotely generated contents, that is by visual objects whose contents bypass the local processor/memory facilities.

This paper presents the application of Switcherland in combination with Oberon System 3 to online multimedia services. We discuss in detail the hardware and software features necessary to efficiently support the management and transport of video and audio data streams, and we describe a client/server environment for online multimedia services. We also give some performance measurements.

2. The Switcherland Distributed System

Current system architectures do not provide a satisfactory platform for online multimedia services. The main shortcoming is the inability of their communication infrastructures to provide quality of service (QoS) guarantees thus making it impossible to process or transfer continuous data such as audio and video streams at a given rate and within specified latency bounds. Further, with today's systems data streams are often forced to take a detour through a processor. A more suitable platform for online multimedia services, however, makes *direct communication* between source and sink devices possible. For example, if a video stream can be transferred directly from a video digitizer to a frame buffer, without involving an intermediate processor, communication bandwidth and processor cycles are saved, and latency and also latency jitter is reduced.

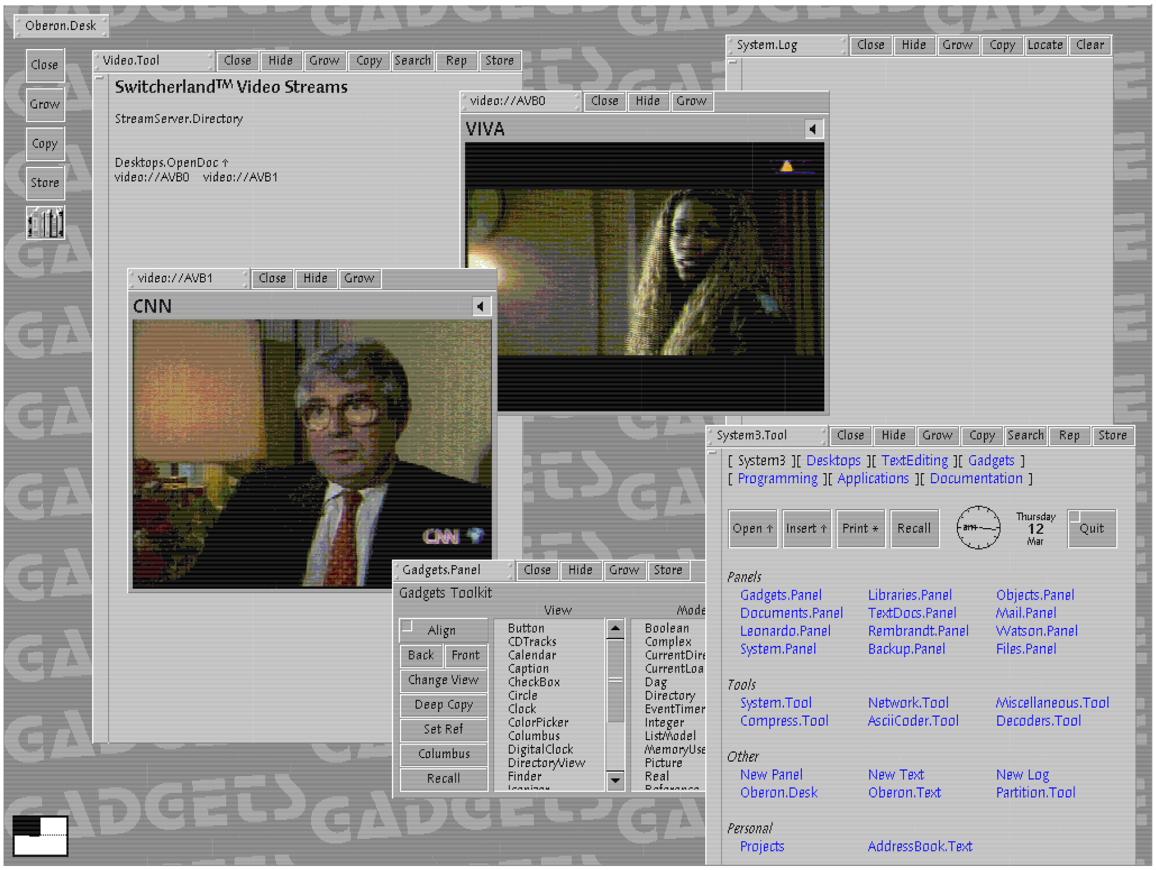

Figure 2: Snapshot of an Oberon System 3 display screen.

We have addressed these shortcomings in the Switcherland project by providing the following features:

- Most importantly, Switcherland's communication infrastructure offers QoS guarantees in the form of connections with reserved bandwidths and bounded transmission delays.
- Multicast is implemented in hardware, an essential feature to efficiently distribute online multimedia contents.
- Any node can be directly accessed by any other node.
- By implementing the basic window management functions directly at the frame buffer level display-oriented data such as video streams can be transferred directly from a source device to a frame buffer without a detour through a processor.

2.1 Bandwidth Guarantees for Multimedia Data Streams

Switcherland realizes a global memory in that all nodes are mapped into one single address space and all communication is translated to *load* and *store* operations giving the programmer a uniform communication model.

The memory operations are transmitted in form of fixed-size 64-byte cells. The resulting data streams are classified as *variable bit rate* (VBR) traffic and *constant bit rate* (CBR) traffic. The two traffic classes differ in the guarantees provided by the switches. For CBR traffic, the switches provide bandwidth guarantees and bounded transmission delays. For VBR traffic, a certain amount of buffering is preallocated in the switches, thereby guaranteeing that cells belonging to this class are never dropped due to overflowing buffers - this is in contrast to traditional networks. Typically, *CBR connections* are used for transferring continuous data and *VBR connections* are used for transmitting data that is generated in bursts, for example, by accesses to disks. Of course, there can be many more CBR and VBR connections than physical connections – our implementation is limited to a maximum of 128 CBR connections and 256 VBR connections per link.

The implementation of the traffic classes is best described by explaining the corresponding flow control schemes: *rate-based flow control* is used for CBR traffic and *credit-based flow control* is used for VBR traffic. In rate-based flow control, the source injects data into the interconnection fabric at a given rate. In credit-based flow control, the availability of buffer space is represented by a credit counter, and the source injects data only as long as credits are available.

Rate-based and credit-based flow control are used in networks such as ATM networks [9][14]. Our implementation differs in that flow control is done end-to-end rather than link-by-link.

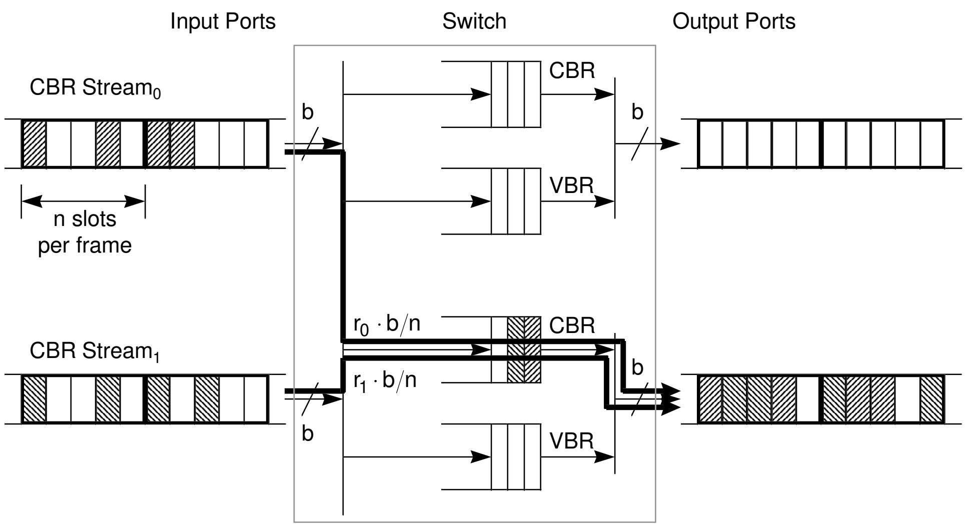

Figure 3: Rate-based flow control.

Rate-Based Flow Control

To describe how rate-based flow control works, we introduce the notion of a *time frame* made up of *n slots*. A slot corresponds to the time needed to transmit a cell. If *b* is the total link bandwidth, bandwidth can be reserved in multiples of *b/n*. Once a bandwidth of $r \cdot b/n$ has been allocated to a certain CBR connection, cells may be sent by the source at a maximum rate of *r* cells every *n* slots. For this purpose, the source is equipped with a pacing mechanism that periodically injects data into the interconnection fabric. The pacing mechanism may use any slots within a frame up to a total of the reserved number of slots. If fewer than the allocated number of slots are used, the unused slots are free for use by VBR cells and, with it, are not wasted. This is useful in combination with a compression method that causes a video or audio stream to be sent over a CBR connection at a variable transmission rate – in this case, a CBR connection is still preferred over a VBR connection to guarantee timely delivery.

The diagram in Figure 3 shows a 2 x 2 port switch with two CBR streams being forwarded from two input ports to one output port. The bandwidth reservations for the two streams are $r_0 \cdot b/n$ and $r_1 \cdot b/n$, respectively. Note that the bandwidth of a link may not be overallocated; applied to the given example $r_0+r_1 \leq n$ must be an invariant. To guarantee that CBR cells are forwarded in bounded time, the switches use separate output buffer queues for CBR and VBR cells and give priority to CBR cells when dequeuing cells. (As a consequence, forwarding delays for VBR cells are variable and unbound.)

Credit-Based Flow Control

Flow control for VBR traffic is credit-based. This is illustrated in Figure 4 and works as follows. A flow-controlled connection uses a full-duplex path between the *client* node and the *server* node. Any operation issued by the client is translated into a *request* and an *acknowledgment*: the client sends a request to the server, which in return sends an acknowledgment back to the client. The bookkeeping required to manage the buffer allocation of a VBR connection is rather simple. At connection set up time, a credit counter in the client is initialized to *c*, whereby *c* represents the number of buffers the client wishes to be reserved in each switch along the path of the connection. Every time a request is sent, the credit counter is decremented, and every time an acknowledgment is received, the counter is incremented. Requests can be sent as long as the value of the counter has not reached zero. With it, there can be at most *c* outstanding requests.

The example in Figure 4 shows a connection with an initial credit $c = 4$. There are three outstanding requests leaving the credit counter at a value of *1*.

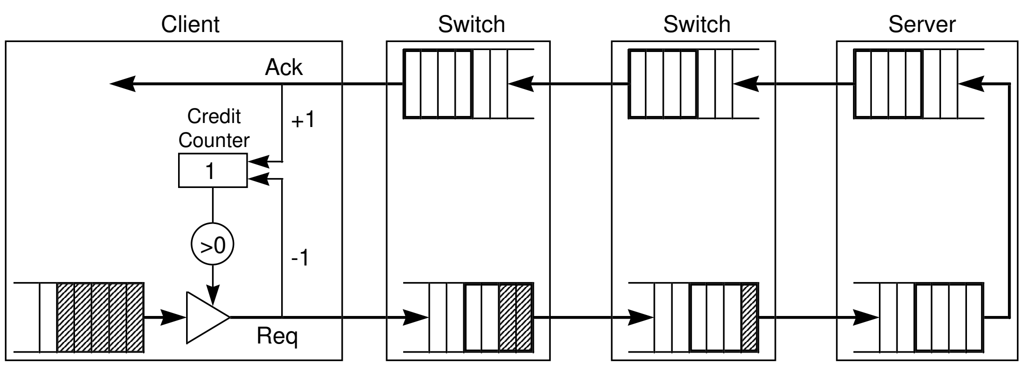

Figure 4: Credit-based flow control.

End-to-End Flow Control

To save buffer space, interconnection networks typically use link-by-link flow control [3][7][8]. Switcherland, however, does flow control for VBR traffic end-to-end. This simplifies the design of the switches considerably since only the nodes and not the switches are responsible for controlling the flow of cells. Of course, as illustrated in

Figure 4 end-to-end flow control in general wastes buffer space since it is not possible to use all buffers. This can only be tolerated if the network diameter is limited. In the case of Switcherland, we restricted the length of any path connecting two nodes to at most ten switches. With it, the round trip time for a connection can be kept short. This, in turn, results in a small window size that requires relatively few buffers in the switches.

Flow control is further simplified since we assume that the nodes can be trusted in the sense that they never exceed their bandwidth and buffer allocations. Therefore, the switches are not required to contain any mechanisms to enforce these allocations.

2.2 Built-in Multicast Support

Since multimedia online services typically connect a data source device with several data sink devices, economic use of the available network bandwidth calls for a multicast capability. Multicast is easily implemented with a bus that is a broadcast medium by its nature and, therefore, can offer the desired feature without much additional support. A broadcast medium, however, does not scale. If scalability is a concern, as is usually the case with distributed systems, switching techniques are an attractive alternative. Scalability, however, comes at the price of higher design complexity as the example of implementing multicast demonstrates: switching techniques rely on point-to-point links rather than multi-point links and, therefore, offer no obvious way to implement multi-point connections. For this reason, switch-based interconnection structures often do not implement multicast connections in hardware but rely on software bundling of several point-to-point connections.

Switcherland implements multicast for CBR traffic in hardware. Though multicast-capable switches have been presented before, we describe a novel and rather simple implementation that makes use of the CBR traffic's properties. To explain how this is done, the implementation of the switch has to be outlined first. Logically, it represents an output-buffered crossbar switch. Physically, the output buffers are implemented with a single shared memory. To be more specific, there is a *cell buffer memory* and a separate *address queue memory*, the latter containing queues with the addresses of the cells in the cell buffer memory. For each output port there is one CBR and one VBR queue. In addition, there is a list of free cell buffers. Figure 5 shows the corresponding arrangement.

Multicast is implemented in that the address of the cell's buffer is written into multiple address queues rather than by storing multiple copies of the cell into the cell buffer memory. This solution is attractive since it does not increase the input bandwidth of the cell buffer memory, which is the critical resource of the design. A complication arises upon freeing a buffer used for a multicast cell since it needs to be determined when the last copy of the cell has left the cell buffer memory. Our solution is to mark the buffer address that goes into the longest queue and free the corresponding buffer by re-inserting its address into the free list when the marked address is taken out of the queue. This works for CBR traffic since the time it takes for a cell to leave the queue is proportional to the length of the queue. The example in Figure 5 shows a cell in buffer 5 that is multicast to output ports A and C.

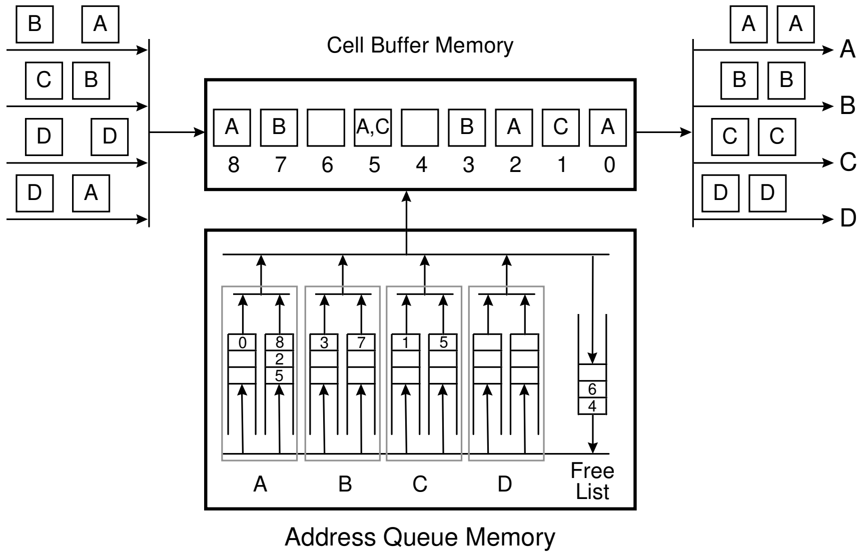

Figure 5: Organization of the Switcherland switch.

2.3 Frame Buffer with Window Management Support

The frame buffer is implemented as an independent Switcherland node that is directly accessible by any other node: it is mapped into the global address space and updated simply by sending write operations.

Direct communication via multimedia streams requires support not only by the interconnection structure itself but also by the individual frame buffer nodes, for example in the form of *clipping* and *windowing*. In principle, these functions can be implemented either in the source device or in the sink device of a stream. In combination with multicast and individual overlapping constellations at the client site, the latter option is a reasonable choice. However, the former option has its benefits as well, because it may save bandwidth, if only one sink device has to be served and parts of a window displaying a video stream are overlapped. This option was the choice in the *DAN clipping nodes* described in [6].

The windowing function offers the abstraction of windows and basically requires an address translation. Instead of addressing pixels absolute within the pixel map, windowing allows pixels to be addressed relative within a window. Thus, a video capture board updates pixels in a virtual window independent of the position of the actual destination window. In this model, the location of a pixel is specified by a window identifier and an address relative to the origin of the window. As shown in Figure 6 the location of the window within the pixel map is obtained by indexing a *window offset* table with the window identifier. The absolute address of a pixel is then given as the sum of the offset and the relative address.

Windowing adds a third dimension in that windows can mutually overlap. When projecting windows onto the two-dimensional screen, it needs to be determined which parts are visible. Hidden parts of a window need to be clipped. We implemented this functionality with the help of a *clipping mask* that stores the identifier of the visible window for every address of the pixel map. Executing a *store* operation now works as follows. First, the absolute address is calculated by the address translation mechanism described. Then, the absolute address indexes the clipping mask to obtain the identifier of the window visible at the corresponding location. Finally, this identifier is compared with the one given as a parameter of the store operation. If they are equal, the pixel is visible and the store operation is executed. Otherwise, the operation is ignored.

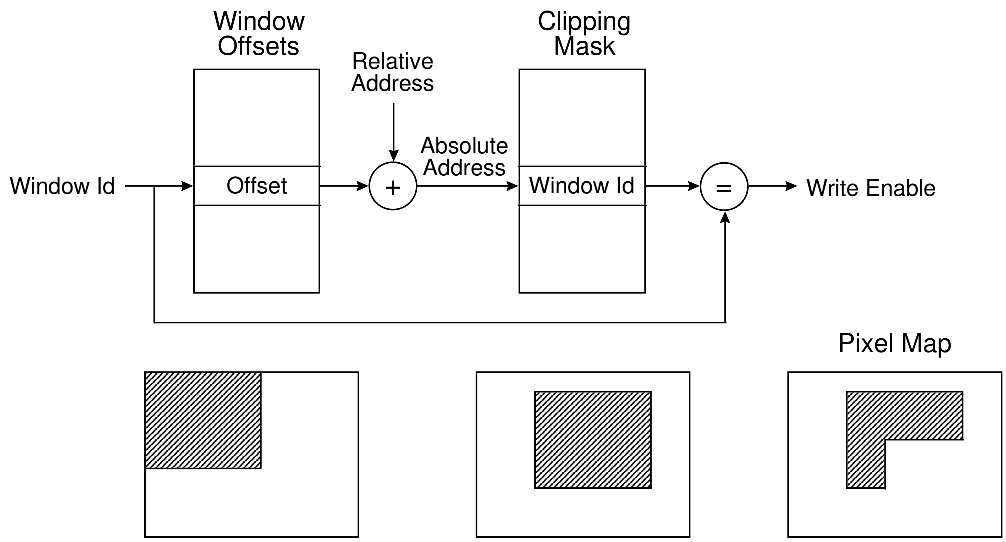

Figure 6: Windowing and clipping.

3. Oberon as an Integrating Display System at the Client Site

Oberon System 3 is used in our project to control, organize and manage at the client site the display of multimedia streams delivered by the Switcherland network. This system distinguishes itself by a highly elaborate concept of a *display space* that, by definition, comprises the entirety of currently visible objects. It is noteworthy that this entirety is a hierarchy of containment of visual objects. In fact, the display space is a visual object itself. More precisely, it is an example of a *container object* that, as such, may contain other container objects like compound documents or GUI panels and elementary objects like character glyphs, buttons and sliders. Containers filled with content objects are called *composite objects*.

In order to understand the specific problems caused by remotely generated objects like multimedia streams, we need some technical background knowledge. Figure 7 shows the internal data structure of a (complex) composite object. It is a dag (rather than a tree) because of the so-called camera views that allow multiple display of one and the same visual component.

By definition, the message protocol for composite objects obeys the principle of *parental control*. In short, this means that messages are never sent to content objects directly but always to their container ("parent") or, looked at it the other way round, that container objects are expected to forward arriving messages to their contents, according to the rules of the protocol. Tightly connected with parental control is the *small-world principle*, basically stating that objects and in particular composite objects are encapsulated worlds by themselves that can be manipulated as a whole under invariance of their inner consistency.

Unfortunately, the small-world principle in its rigorous form can no longer be maintained in combination with remotely generated contents. The example of moving a composite object containing a remotely generated component from one location in the display space (which is the root of all container objects) to another one may serve as an example. This is the sequence of events happening behind the scenes when an interactive user wants to move the CNN panel in Figure 2 to a different location:

(1.) A mouse-event message is sent to the display space and forwarded to CNN.
(2.) CNN takes control, tracks the mouse and registers the desired new location.
(3.) CNN broadcasts a *remove* message to the display space to remove itself at the old location. The effect of this message is a redrawing of the affected objects.
(4.) CNN broadcasts a *redraw* message to the display space to redraw itself at the new location.

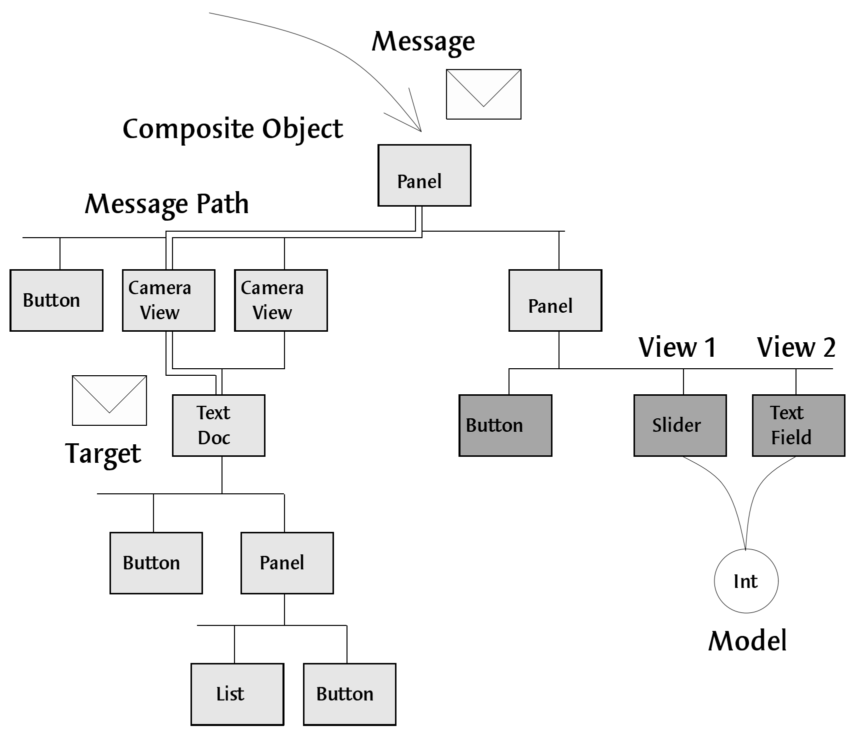

Figure 7: A composite visual object.

Taking a refined look at the implementation of the *redraw* operation, we first notice that *redraw* requests are broadcast in the display space, because they typically demand computation of a new clipping mask, what is conveniently done incrementally while the message is travelling down the hierarchy along the context path. As far as redrawing itself is concerned, we now recognize a fundamental difference between locally and remotely generated contents. Local contents are simply bitblock-transferred or re-generated by raster operations using the clipping mask. In contrast, remotely generated contents have to be redirected to a different location in the display memory and, possibly, clipped. Thanks to the architectural support provided by Switcherland frame buffers as explained in the previous section, redirection and clipping are simple: adjust the offset of the content window and update the clipping mask table accordingly.

We can now characterize our approach and solution as based on (a) tagging remotely generated objects in the display space and (b) augmenting the message protocol, so that containers are obliged to forward critical messages to their tagged components. Obviously, to make this work correctly, containers need to "inherit" the *remote*-tag from their contents.

Another principal problem are the already mentioned camera views in combination with remotely generated contents. A camera view in Oberon is a (possibly partial) view of some visual object, for example a picture or drawing. In the case of local contents, multiple camera views on the same object may well be present simultaneously in the display space. This, however, is no longer true for remotely generated contents under the current architecture, because streams may not be multiplied in the same frame buffer. The current implementation always directs a stream to its latest view.

4. The Resulting Multimedia Client/Server Environment

We envisage a Switcherland environment whose sole purpose is to provide multimedia programs. The main hardware constituents are data sources, connections, switches and clients. Many different forms of sources are conceivable, among them cameras and microphones, TV receivers and CD players. Some of them require a selection among a set of potentially available channels.

Taking an imaginary snapshot of the environment, we see a possibly large number of video and audio streams flowing from their source to one or several clients. The operations "show/stop program P on client station C" or, more technically, "open/close stream P from source S to client C's frame buffer" define the basic state transitions. Obviously, a coordinating authority is needed for the handling of such requests. For that purpose, we have implemented a central *program server*, again a PC running Oberon with a connection to the Switcherland hardware via a PCI adapter. Its main responsibilities are book keeping, routing and multiplexing of multimedia streams. Although we have not done it, the server could easily be upgraded by an authorization and access control on a per-client and per-program basis.

It is important to realize that the server is used momentarily only to answer client requests and is not further involved in the actual data transfer. Hence, the non-scalability of the central server architecture is not an issue in this case. In detail, the client-oriented services of the program server are:

(1.) Enumerate all currently available streams
(2.) Open a desired stream for a client
(3.) Close a specific stream to a client

Streams are identified to the program server by a qualified name of the form *SourceId.StreamId*, for example: *TV.Video* or *TV.Audio*. Obviously, a client station can display many different video streams but play back just one audio stream (that of the program focused on) at any time.

In addition to client-oriented operations, the program server offers a central control interface for convenient maintenance of different sources. The provided commands are:

(4.) Select a desired channel for a certain source
(5.) Start/stop stream
(6.) Define the pixel resolution of a (video) stream

Figure 8 is a special case of Figure 1. It shows a system consisting of a frame grabber FG acting as a video source, two clients and a program server. Server and clients use PCs connected to Switcherland switches via PCI adapters (PCIA). At the beginning, there is a video stream V_1 originating at the frame grabber FG and ending at the frame buffer FB_1 of $Client_1$. $Client_2$ now sends a request to the program server to open stream V_1 and send it to frame buffer FB_2. In response, the program server updates the switches' routing tables $RTable_0$ and $RTable_2$ by modifying the entries indexed by the identifier of connection C_1, which is used for transporting stream V_1. An entry contains a bit vector that identifies the output ports the stream has to be forwarded to.

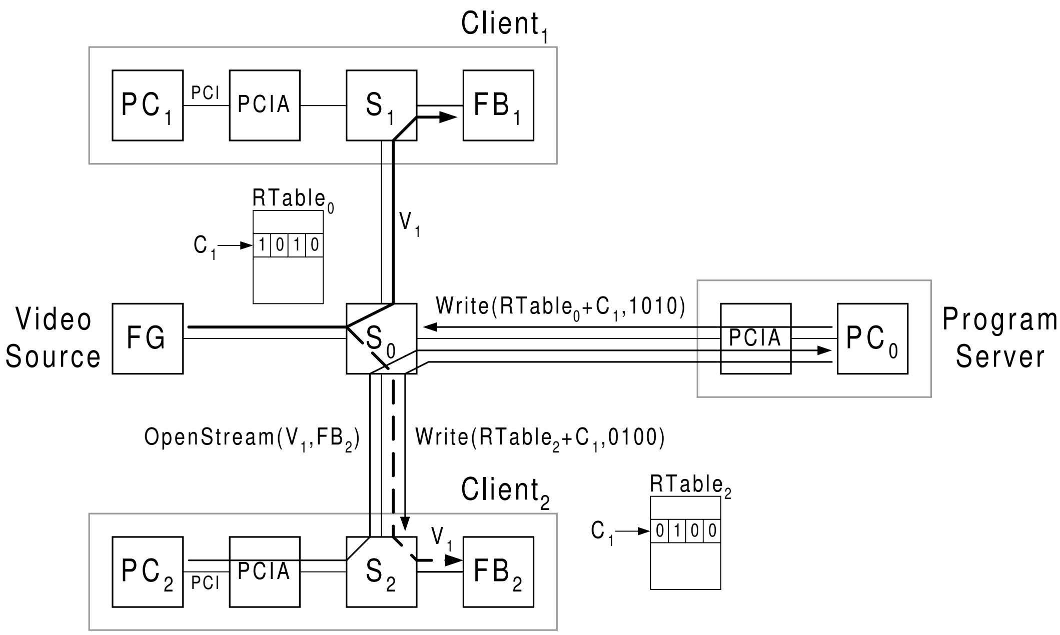

Figure 8: Client/server architecture.

5. Switcherland System Performance

Our environment currently consists of seven Switcherland workstations each containing a switch, a frame buffer and a PC. PCs are connected via PCI adapters. In addition, there is a PC running the program server. Workstations and server are interconnected by an additional six switches.

We have implemented a 4 x 4 port switch. Each port connects to a 265.625 Mbit/s full-duplex serial link compatible with the physical layer of the FibreChannel standard. Not considering the overhead caused by the 8B/10B encoding scheme [10] used on the links, a 4 x 4 port switch offers an aggregate bandwidth of 0.85 Gbit/s. The latency is 1.5 µs per switch, which is at least an order of magnitude faster than the latency measured for ATM switches [1].

The frame buffer stores pixel values as true-color 24-bit RGB values in a 1024 x 768 pixel map. The pixel map can be accessed at the full link bandwidth. Clipping mask and window offset table of the frame buffer work with 8-bit window identifiers, thus, a total of 256 windows can be supported. While the number of video windows in our system is mainly limited by the available link bandwidth, many other systems and, in particular, commodity systems are restricted to one or only a few video windows. Further, these systems are often limited in that video windows cannot be overlapped at all or display update performance is severely degraded when windows overlap.

The following two sections investigate the bandwidth available to update the display screen. We distinguish between VBR and CBR connections: CBR connections are used to transmit video data from a video source device to the frame buffer and VBR connections serve to transmit control data as well as textual and graphical data from a PC to the frame buffer.

5.1 VBR Data to the Display

Figure 9 shows the amount of bandwidth available when an application writes VBR data to the frame buffer. The bandwidth was measured as a function of the block size of the written data. Measurements were done for connections with different amounts of credits. The graphs show that a throughput close to the full link bandwidth can be achieved if a sufficient number of credits is available and large enough block sizes are chosen. The performance degradation for smaller blocks can be explained by overhead caused by the driver.

The measurement setup used a frame buffer directly connected to a host. For a path containing the maximum number of ten switches another 13 credits have to be added to achieve performance numbers similar to the ones presented. This number is given by the latency added by the switches.

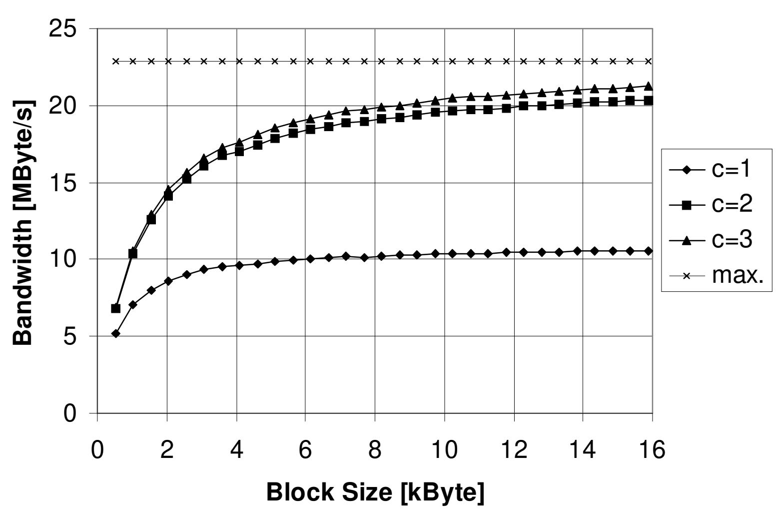

Figure 9: VBR throughput.

5.2 CBR Data to the Display

Figure 10 shows the update rates measured as a function of the size of the written block. Measurements were done for different bandwidth allocations. Again, for smaller block sizes the allocated bandwidth cannot be fully utilized due to driver overhead.

The maximum bandwidth available for transmitting CBR video streams to the frame buffer is given by the link capacity. Taking the overhead given by the link encoding as well as the cell headers into account, data can be written into frame buffers at a maximum rate of 183 Mbit/s. When transmitted, pixel values are packed as 32-bit values giving a maximum rate of 5.7 MPixel/s. This rate corresponds to about 12 video streams displayed at a resolution of 160 x 120 pixels per frame and 25 frames per second, or 3 video streams displayed at a resolution of 320 x 240 pixels per frame and 25 frames per second.

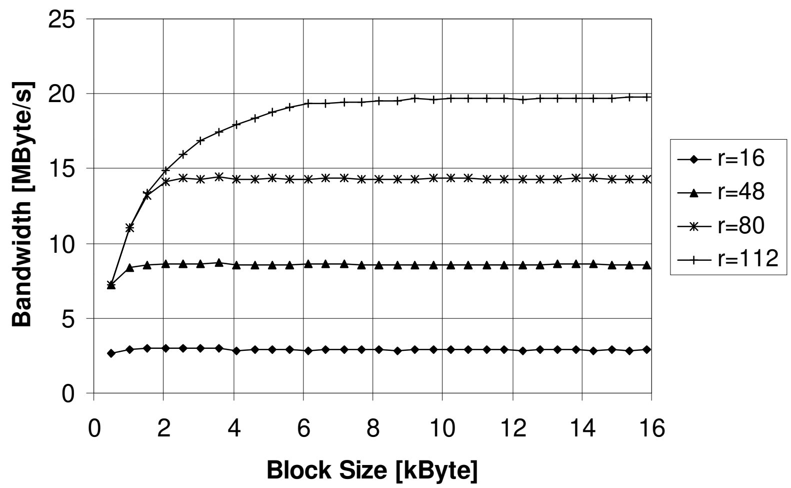

Figure 10: CBR throughput.

6. Related Work

Architectures and frameworks for processing multimedia data are the topic of many research efforts. Their focus typically is the timely manipulation of continuous data, which is a prerequisite for a multimedia system. As we have shown, certain types of multimedia systems can further benefit from a communication infrastructure that enables data transfers directly between peripheral devices. To make this possible, modifications to their hardware and software are necessary. The literature, however, does not cover this aspect sufficiently.

Medusa [17] is a networked multimedia environment based on ATM technology. Much like Switcherland nodes, peripherals including cameras, displays and audio input/output are separate components, each independently connected to the network without being attached to a specific workstation. The software system uses a *peer-to-peer architecture* based on point-to-point connections that allows source devices to send data directly to sink devices.

Similar to Medusa, the VuSystem [12] uses ATM technology to interconnect general-purpose workstations and network-based multimedia devices. It provides a programming system for multimedia applications that perform complex computations on video and audio data. Applications are mapped onto a *media-flow architecture* consisting of three modules: source, filter and sink. Communication between modules is again point-to-point.

Medusa and the VuSystem do not consider the delivery of a data stream to multiple sink devices. In this respect, the Nemesis project [11] provides an infrastructure better suitable for this task. Nemesis is an operating system that provides QoS guarantees to applications. The *Nemesis Device Driver Architecture* makes a clear separation of control-path and data-path operations. In [2] Barham gives an example of a frame buffer driver and corresponding window system where all rendering is performed by the client and updates to the frame buffer are performed directly by the client at a rate determined by per-connection QoS parameters. The frame store used [13] provides clipping and windowing mechanisms similar to the ones described here. While Nemesis allows several clients to share a frame buffer, no control mechanism is described that allows a stream to be multicast, that is, to be shared by several frame buffers.

7. Conclusion

Direct communication between multimedia peripherals is the key to an efficient integration of online multimedia services into computer networks. The necessary changes and additions to existing system architectures have been described in this paper: on the hardware side, guaranteed transmission bandwidth for audio/video streams, multicast support, and globally accessible display frame buffers are needed; on the software side, the management of visual objects has to cope with remotely generated contents.

A corresponding system has been successfully implemented. It allows the delivery of high-quality audio and video to the user at no extra cost in respect to processing power. Displaying video is done with the help of visual objects that can be manipulated like traditional local objects, in particular, that can be moved around and overlapped arbitrarily.

Acknowledgements

Patrick Saladin has adapted System 3 to Switcherland. Daniel Scherrer has implemented the program server. Michaela Blott, Erwin Oertli and Peter Ryser have built the Switcherland switch and nodes used for this project.

References

[1] G. Babic, A. Durresi, R. Jain, J. Dolske, S. Shahpurwala: *ATM Switch Performance Testing Experiences*, ATM Forum/97-0178R1, April 1997, http://www.cis.ohio-state.edu/~jain/atmf/a-0178r1.htm.

[2] P. Barham: *Devices in a Multi-Service Operating System.* Technical Report 403, University of Cambridge, October 1996.

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su: *Myrinet: A Gigabit-per-Second Local Area Network*. IEEE Micro, vol. 15, no. 1, February 1995, pp. 29-36.

[4] H. Eberle: *Switcherland: A Scalable Interconnection Structure for Distributed Systems*. Journal of System Architectures, Elsevier, vol. 44, nr. 2, 1997, pp. 227-240.

[5] J. Gutknecht: *Oberon System 3: Vision of a Future Software Technology*. Software - Concepts and Tools, Springer, February 1994. vol. 15, nr. 1, 1994. p 26-33.

[6] M. Hayter, D. McAuley: *The Desk-Area Network*. ACM Operating Systems Review, vol. 25, no. 4, October 1991, pp. 14-21.

[7] R. Horst: *TNet: A Reliable System Area Network*. IEEE Micro, vol. 15, no. 1, February 1995, pp. 37-45.

[8] M. Katevenis: *Telegraphos: High-Speed Communications Architecture for Parallel and Distributed Computer Systems*. Technical Report 123, Foundation for Research and Technology, Heraklio, Crete, 1994.

[9] H. Kung, R. Morris: *Credit-Based Flow Control for ATM Networks*. IEEE Network, 40-48, March/April 1995.

[10] C. Jurgens, *FibreChannel: A Connection to the Future*. IEEE Computer, vol. 28, no. 8, August 1995, pp. 88-90.

[11] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. Hyden: *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications*. IEEE Journal on Selected Areas in Communication, vol. 14, nr. 7, September 1996, pp. 1280 -1297.

[12] C. Lindblad, D. Tennenhouse: *The VuSystem: A programming System for Compute-Intensive Multimedia*. IEEE Journal on Selected Areas in Communications, vol. 14, nr. 7, September 1996. 1298-1313.

[13] I. Pratt: *A Key Based Framestore for the Desk Area Network*. In ATM Document Collection 3 (The Green Book), University of Cambridge, October 1995, pp.1 4.

[14] K. Ramakrishnan, P. Newman. *Integration of Rate and Credit Schemes for ATM Flow Control*. *IEEE Network*, 49-56, March/April 1995.

[15] N. Wirth: *The Programming Language Oberon.* Software - Practice and Experience, vol. 18, nr. 7, 671-690, 1988, pp. 671-690.

[16] N. Wirth, J. Gutknecht: *The Oberon System.* Software - Practice and Experience, vol. 19, nr. 9, 1989, pp. 857-893.

[17] S. Wray, T. Glauert, A. Hopper: *Networked Multimedia: The Medusa Environment*. IEEE Multimedia, vol. 1, nr. 4, Winter 1994, pp. 54-63.

[18] G. Krasner and S. Pope: *A Cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, vol. 1, nr. 3, August 1988, pp. 26-49.