The Multimedia Multicast Channel

J. Pasquale and G. Polyzos and E. Anderson and V. Kompella

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego

Abstract. The Multimedia Multicast Channel (or simply "the Channel") provides a "television broadcast channel" programming abstraction for a group of distributed processes which communicate multimedia information such as video and audio streams in a dissemination-oriented fashion. By dissemination-oriented, we mean that there is a single multicasting source, with a varying number of destinations "tuned in" to a Channel, and that there is a very loose coupling between the source and the destinations.

1 Motivation

The Channel is based on a communication paradigm which is a major departure from more traditional ones. In this paradigm, the source and the destinations are very loosely coupled in their control and data exchange interactions. In general, the source's main concern is to push various media streams into the Channel; where they end up (i.e. who are the actual receivers), or how they are used (e.g. what will specific receivers extract from any or all of the streams), may be of little concern. A destination's main concern is what to extract from the Channel; the Channel is viewed as offering multiple media streams, some or all of which are of interest.

Perhaps the most unique feature of this communication paradigm is that it is not unusual for the forms of the media streams, generated by the source and required by the different destinations, to be quite different. For example, the source may generate HDTV-quality video and CD-quality audio, whereas some destinations can only use NTSC video and its associated audio, while other destinations can use only audio and no video. Indeed, every destination may have very different and independent requirements. Therefore, it is expected that the destinations will individually tailor, to a high degree, what streams are actually received. These streams may be a subset of the source streams, or new ones computed from the source streams. We encourage the use of component coding (see Discussion section) at the source to facilitate tailoring.

We believe this communication paradigm is highly appropriate for multimedia distribution services required by a large class of multimedia multipoint applications. A prime example is "video distribution" [GM91, Sinc90], as in Cable Television systems where a single source generates video (and associated audio) distributed to a large set of receivers who generally have little or no interaction with the source. Another application is video conferencing

[Casn90, Enso88, VZSR91], where each member is both a source and receiver, and would require one of our Channels per source to support base-level audio/video distribution. However, video conferencing also requires other control mechanisms which are outside the scope of what is provided by the Channel.

More generally, one can distinguish between higher-level application-specific control mechanisms (e.g. acquire/release floor-control, voting, "change the TV channel"), and lower-level media-oriented control mechanisms (e.g. for a media stream, modify resolution, granularity, or intensity; for multiple media streams, mix or synchronize). Media-oriented control does not imply (explicit) control between source and destination, and is generally useful in most multimedia applications; consequently the Channel supports media-oriented control (through its filter mechanism). All other required control, particularly control between source and destination, is expected to be provided by the application itself (perhaps through libraries/toolkits for common source-destination control functions, e.g. floor control).

In this extended abstract, we describe the general design of the Channel, followed by a discussion. Due to space limitations, we have moved the section on past work and the section on the system call interface to appendices placed at the end of this document.

2 The Channel Design

2.1 Channels

A "channel" is single-source multiple-destination dissemination-oriented group communication abstraction. A channel transports data of different media types on different streams. A source can create a channel, specifying the types of data it expects to send and who the potential recipient users are through an access control list. Any process on the list can dynamically "tune in" to the channel, at which time it may receive on a channel port any of the streams generated by the source.

2.2 Ports

A port is a process's control and data access point of a channel. Ports are opened and closed, and when open, streams associated with the port can be enabled, allowing the flow of data.

2.3 Streams

A port provides access to one or more "streams" generated by the channel source. Streams travel in one direction, from source to destination. A stream is typed, with different types corresponding to different media or media components. For example, a source may generate the following streams:

audio left, audio right, video luminance, video chrominance

The first two streams are both of type "audio". The video luminance and chrominance streams correspond to two different types of streams.

Each stream has a priority, which describes the stream's importance/precedence relative to other streams on the same port. The priority may also describe hierarchical relationships between streams. For example, video luminance and video chrominance are hierarchically related, with the former taking precedence over the latter. Priority is useful for congestion control and scheduling policies.

There are two additional stream parameters which affect the position of stream segments (described next): translation and scaling (described below).

2.4 Segments

A stream is actually a sequence of "segments". These are treated as atomic units of information at the system call interface: a user process will always send and receive entire segments, no partial segments are sent or received. The segment is also the smallest unit of information operated on by filters, described below.

A segment has a position number, which describes the segment's position in the stream relative to the beginning of the stream, which is always position 0. One useful (but not necessary) interpretation of the position is as a position in time (relative to the time at the beginning of the stream).

A segment also has a length, which is the logical length of the segment within the stream. The length is not (necessarily) the same as the segment's physical size in bytes. For example, a segment for a video stream may be a compressed frame, which has a size in bytes that is generally smaller than its logical length which could be interpreted as the size of the uncompressed frame in bytes.

The main purpose of the segment's position (and length) is to allow "alignment" of multiple streams. To align multiple streams, a "channel position number" is calculated for each stream as follows: a segment's stream position number P is scaled and translated by the stream's scaling and translation parameters, S and T, resulting in a "channel position number" $P' = T + S \times P$. (The segment's channel length L' is computed from the its stream length L as follows: $L' = S \times L$.) Segments from different streams are said to be "co-located" if their channel position numbers are the same. Multiple-stream alignment makes sophisticated filters possible by allowing computation to operate on co-located segments.

2.5 Filters

A receiving process can filter a set of streams from the same port to obtain a new filtered stream. A filter's execution is triggered upon the arrival of a segment from one of a specified set of streams, or after a programmable timeout interval (measured since the last execution began). The filter can queue the segment for later processing, pass it through, or do both by queueing a copy and passing it through.

When a filter has collected enough segments (governed by filter code), it may then operate on the segments to produce a new segment. The filter uses the channel position numbers and channel lengths for each segment to determine which are the corresponding segments on which to operate. Some standard filters include AVG (take the average of streams), and μ-LAW-ADD (mix μ-law formatted audio streams).

Programmer-defined filters are also supported. The programmer supplies a function whose execution is triggered as described above, and is called with information about the segment triggering its execution (or an indicator that a timeout has occurred).

When a filter is installed, it may propagate into the network on connections created by the channel and execute as close to the source as possible if it is efficient to do so. For instance, if a destination is only interested in the average of two streams, but not the two streams themselves, then it would be inefficient to transmit the two streams only to discard them after they are averaged. To insure protection wherever filters are located, filters execute in protected domains with suitably limited instruction sets, and with safeguards against CPU hogging.

3 Discussion

Our main goal is to support dissemination of information from a source to multiple destinations. The loose coupling between the source and destinations suggests an open-loop approach to flow control. With multiple destinations, closed-loop control is difficult and cumbersome. For example, flow control of connections to multiple destinations results in the slowest receiver impeding the progress of the others. Our open-loop approach is necessary because of the different properties and requirements of real-time audio and video (versus traditional data) transmission, the multipoint (versus single receiver) communication mode, and the intended final user (typically a human in the case of continuous media, versus a program – most of the time – in the case of traditional data).

Such an approach is particularly attractive when one considers media transmitted in the form of independent or hierarchically dependent components (such schemes are referred to as hierarchical, component, layered, or sub-band coding). We encourage component coding because we anticipate that user terminal capabilities and network access characteristics will cover a very wide spectrum, and we expect users to tailor the media they receive from the source to their capabilities. With multiple users trying to impose their views on the specific format of the information disseminated by the source, the problem quickly becomes intractable. Thus, we opted for a design where a source provides the highest quality (possible or desired) of the media/signal in independent components, which can easily be used even by less sophisticated devices.

At a lower level, a segment, with its size defined by the application, is considered a relatively independent piece of information. It is advantageous for the network protocol software to treat the segment as the data unit of manipulation [CT90]. Applications are expected to recover gracefully from loss of segments (at least for some media components). Therefore, cooperation between the network system software and the operating system dictates that segments be transported atomically through the protocol stack and the network. Even though this is not a requirement, the approach taken here would suggest that if any part of a segment is damaged or lost, the whole segment should be discarded. Our emphasis on not necessarily reliable data transmission stems from the following observations:

1. for "real-time" continuous media, smooth progress is more important than (complete) reliability
2. typically, there is no time for re-transmission of corrupted data (e.g., on transcontinental links).

Channel performance and semantics depend on the quality of service (QoS) provided by the underlying system. Each stream data type dictates what QoS is expected from the network and I/O system. How the underlying system makes provisions to implement quality of service demands is outside the scope of this paper, but is an important issue – see [FV90, Zhan91] for various schemes. In particular, we will address the ramifications of our open loop approach on the underlying system support for quality of service in the full version of this paper.

4 Conclusions

We have described the design for the Multimedia Multicast Channel, which is based on a dissemination-oriented communication paradigm. It is useful in supporting multimedia multipoint applications. It provides the base-level support for media distribution and media-oriented control, leaving the responsibility for high-level group control to the application. Through the use of channel filters, destinations can tailor the media streams they receive. The loose coupling between source and destinations suggests an open-loop approach to flow control.

5 Appendix 1: Past Work

Multicasting has received considerable attention lately because of the interest in collaboration technologies. However, little attention has been paid to appropriate system support for this mode of communication. Most of the work in this area is revolving either around low-level communication protocols or groupware applications, which typically simulate multicast through a series of unicasts. However, the support for dissemination-oriented communication paradigm is beginning to receive attention [Cher92].

The pioneering work on IP multicasting [DC90] is fueling some of the current protocol work, which is based on a group communication protocol, the Internet Group Management Protocol (IGMP) [Deer89]. This protocol specifies how multicast groups can be formed and managed with little change to the basic infrastructure of IP. The two major changes to the implementation of IP to accommodate multicasting are the provision of IP multicast addresses and the IGMP protocol described in [Deer89]. Multicast groups are created by individual hosts explicitly joining the groups, i.e., using IGMP to include them in the address list corresponding to a multicast address. The actual multicast routing is done by routers that learn the shortest path routes to the destinations. These routers then forward multicast packets along those shortest paths [DC90].

Recent experiments using IP multicasting include the telecasting of the last IETF meeting. Live audio and slow scan black-and-white video from the meeting was distributed to hosts that wanted to be included in the multicast group. Tunneling [WPD88] was used to send data over portions of the Internet that did not support multicast routing.

An alternative network layer protocol in the Internet suite, specially designed for continuous media transmission, is ST-II [Topo90]. This experimental protocol supports multicasting and resource negotiation appropriate for continuous media, but the protocol itself does not include specific implementation mechanisms, e.g., for multicast route set-up or network resource reservation (instead, such mechanisms must be provided outside of the protocol).

The multicast routing algorithm described in [KPP92] optimizes network performance for continuous media, making it a strong choice for Channel support. It constructs a source-rooted multicast tree which efficiently uses bandwidth while bounding delay between the source and all destinations. Other work on optimization of multicast tree set up, some of it in the context of multimedia communications, can be found in the references in [?].

Hierarchical coding has been studied extensively in the area of signal processing. It has recently been recommended as a potentially effective mechanism for congestion control for high-speed networks carrying digital continuous media [KV89, Ghan89, PPEA91, FPP92, Shac92]. However, we are not aware of any other operating system or network system software level support for layered coding, in particular, in relation to multicasting.

6 Appendix 2: System Call Interface

```
c = CreateChannel (ac, spec)
int c;
ACCESS_CONTROL ac[];
STREAM_SPEC spec[];
```

Create channel and return a channel number, `c`, for identification by potential users of this channel. `ac` is a zero-terminated array describing a set of potential users, not necessarily through explicit enumeration. `spec` is a zero-terminated array where each entry describes a sourced stream that includes: type (e.g. VIDEO), format (e.g., μ-law), maximum size of a segment, priority, position translation parameter, position scaling parameter, and a string describing additional info (e.g., left or right channel if stereo audio). If $c < 0$, an error occurred.

```
p = OpenPort (c, spec)
int p, c;
STREAM_SPEC *spec[];
```

Open a new port to channel `c` (which must have already been created), and return a local port identifier `p`. Also return a pointer to a stream specification array, which is a copy of the structure used to create the channel. If $p < 0$, an error occurred.

```
ClosePort (p)
int p;
```

Close a previously opened port `p`.

```
EnableStream (p, s, sd, numsd, buf, bufsize)
int p, s, numsd, bufsize;
struct segment_descriptor sd[];
char *buf;
```

Enable segments to be sent to or received from stream `s` of port `p`. Segments are placed in a buffer area `buf` of size `bufsize` (in bytes) which is managed by the kernel as a circular buffer. Information describing segments in the buffer area is maintained in the segment descriptor array `sd`.
Segment descriptors are structured as follows:

```
struct segment_descriptor {
        char *seg;      /* pointer to segment in buf */
        int size;       /* size of segment in buf */
        int pos;        /* logical position of segment in stream */
        int len;        /* logical length of segment in stream */
};
```

`seg` is a pointer to the segment in the buffer. If `seg` = 0, then there is no segment pointed to by `seg` and the segment descriptor is free. Initially, all segment descriptors should be marked free by having the `seg` = 0. For a destination user process receiving a stream, the kernel (asynchronously) places segments into the `buf` area and fills in the next free segment descriptor (the segment pointer will then be non-zero). After the destination reads (or does whatever it wishes) with the segment, it sets `seg` = 0 allowing the kernel to reuse the segment descriptor. For a source sending a stream, the roles of the user process and kernel are reversed: the user process places segments into the `buf` area and fills in the next segment descriptor marked free. After the kernel is done with the segment, it sets `seg` = 0 allowing the source user process to reuse the segment descriptor.
`size` is the size of the segment in the buffer area pointed to by `seg`.
`pos` is the logical position of the segment in the stream.
`len` is the logical length of the segment in the stream.

```
DisableStream (p, s)
int p, s;
```

Disable segments being sent to or being received from stream `s` of port `p`.

```
fs = FilterStreams (type, func, p, s, ns, timeout)
int fs, type, (*func)(), port, s[], ns, timeout;
```

Filter a set of streams belonging to port `p` and specified in array `s` of length `ns` and return a new filtered stream `fs`. The filter executes whenever a segment of one of the streams in `s` encounters it, or after a timeout interval.
There are a set of well-known filters named by `type`. To allow for programmer-defined filters, if `type` = USER, then the filter's code is defined by `func`, a user-supplied function, which is passed by the calling process.
A filter is called (by mechanisms internal to the channel upon segment arrival or timeout interval expiration) as follows:

```
(*func) (s, sd)
int s;
struct segment_descriptor *sd;
```

`func` is called with a stream identifier s and segment descriptor `sd` from which it can determine the segment address in memory, its size, and its stream position and stream length. If `s` = NULL (no stream), then `func` was called due to a timeout, and `sd` is undefined.

References

[Casn90] S. Casner, K. Seo, W. Edmond, C. Topolcic: N-Way Conferencing with Packet Video. 3rd International Workshop on Packet Video, Morristown, NJ. (March 22-23, 1990)

[Cher92] D. Cheriton: Dissemination-oriented Communication Systems. Distinguished Lecturer Series, Department of Computer Science and Engineering, University of California, San Diego, (April 6, 1992)

[CT90] D. D. Clark and D. L. Tennenhouse: Architectural Considerations for a New Generation of Protocols. Proc. ACM SIGCOMM '90 (Sep 1990) 200–208

[Deer89] S. Deering, "Host Extensions for IP Multicasting," ISI, RFC 1112, Network Information Center, SRI International, Menlo Park, CA, August 1989.

[DC90] S. Deering and D. Cheriton, "Multicast Routing in Internetworks and Extended LANs," ACM Transactions on Computer Systems, vol. 8, pp. 85-110, May 1990.

[Enso88] J. R. Ensor, S. R. Ahuja, D. N. Horn, and S. E. Lucco, "The Rapport Multimedia Conferencing System - A System Overview," IEEE Computer, April 1988, pp. 52-58.

[FPP92] D. Ferrari, J. C. Pasquale, and G. C. Polyzos, "Network Issues for Sequoia 2000," Proc. IEEE Compcon Spring '92, San Francisco, CA, February 24-28, 1992

[FV90] D. Ferrari and D. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," IEEE JSAC, vol. 8, pp. 368-379, April 1990.

[Ghan89] M. Ghanbari, "Two-Layer Coding of Video Signals for VBR Networks," IEEE JSAC, vol. 7, no. 5, pp. 771-781, June 1989.

[GM91] R. Gusella and M. Maresca, "Design considerations for a multimedia network distribution center," Proceedings 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Germany, November 1991.

[KV89] G. Karlsson and M. Vetterli, "Packet Video and its Integration into the Network Architecture," IEEE JSAC, vol. 7, no. 5, pp. 739-751, June 1989.

[KPP92] V. Kompella, J. Pasquale, and G. Polyzos, "Multicasting for Multimedia Applications," Proc. Infocom'92, pp. 2078-2085, Florence, Italy, May 1992

[PPEA91] J. Pasquale, G. Polyzos, E. Anderson, K. Fall, J. Kay, V. Kompella, D. Ranganathan, S. McMullan, "Network and Operating Systems Support for Multimedia Applications," TR CS91-186, UCSD, March 1991.

[Shac92] N. Shacham, "Multipoint Communication by Hierarchically Encoded Data," Proc. Infocom'92, pp. 2107-2114, Florence, Italy, May 1992.

[Sinc90] W. D. Sincoskie, "Video On Demand," Proc. IEEE Globecom '90, San Diego, 1990.

[Topo90] C. Topolcic, Ed., "Experimental Internet Stream Protocol, Version 2 (ST-II)," RFC1190, Network Information Center, SRI International, Menlo Park, CA, October 1990.

[VZSR91] H. M. Vin, P. T. Zellweger, D. C. Swinehart, and P. V. Rangan, "Multimedia conferencing in the Etherphone environment," IEEE Computer, vol. 24, no. 10, pp. 69-79, October 1991.

[WPD88] D. Waitzman, C. Partridge, and S. Deering, "Distance Vector Multicast Routing Protocol," RFC 1075, Network Information Center, SRI International, Menlo Park, CA, November 1988.

[Zhan91] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," ACM Trans. on Computers, vol. 9, no. 2, pp.101-124, May 1991.