Dynamic Data Path Reconfiguration

Carsten Griwodz[1], Michael Zink[2]

[1]griff@ifi.uio.no
University of Oslo - Department of Informatics
Gaustadalléen 23 - 0371 Oslo, Norway

[2]michael.zink@kom.tu-darmstadt.de
Darmstadt University of Technology - Industrial Process and System Communications (KOM)
Merckstr. 25 - 64283 Darmstadt, Germany

I. Introduction and Motivation

When trying to build an audio/video (AV) streaming infrastructure one realizes that some functionality is needed in several parts of the application. E.g., in case of video server and client, protocols like RTP/RTCP, RTSP and SDP are needed in both parts. Therefore it is appropriate to implement these functions in a reusable manner and to create a well-defined and documented API for each module. To use the infrastructure in projects with non-research groups, it is necessary to support several decoders and several video servers for the diverse encoding formats (e.g. H.263, MPEG-1, QuickTime). These come with different APIs, leading to an adaptation effort whenever a new library is integrated.

While such abstractions are typical for streaming applications, a generic structure like that of the JMF [1] is rarely found in free, open source systems. Existing approaches implement either hard-coded sequences, or they consider frameworks that allow the specification of an end-to-end behavior for complex multimedia systems. In the latter kind of systems, functionality is described at the level of cooperating distributed components [2, 3]. It is typical for such frameworks to consider networking as a component that is also under the control of the framework.

In an environment that ensures interoperability by specifying protocols (such as the RTSP streaming environment), we prefer a local approach. The control of our framework extends only over a single machine and RTSP is used explicitly for communication.

Due to the interaction of RTP and RTCP, and the possibility of receiving data from several sources at a single port, a directed, non-cyclic graph of modules is an appropriate streaming model. In case of RTP, an infrastructure is appropriate only if dynamic reconfiguration of the data path is supported by the modules as well as the controlling framework. A packet that arrives at an interface from an unexpected sender must be handled in an application-defined way: it may be appropriate to discard the packet, to assign it to a default path, or to create an additional stream for special processing.

Dynamic reconfiguration must also be supported in the implementation of a proxy cache server: it is necessary to handle user interaction if that cache implements a write-through mode. The client receives data from the origin server through the proxy cache, which writes the data to disk and forwards it to the client as well. If the client pauses and the application decides to continue the caching operation, the trunk of the graph that forwards data to the client must be cut, while the trunk that stores data on disk must be maintained. If the client resumes viewing, the application must create a new graph, which retrieves the data from the cache.

We explain our design decisions that are derived from these requirements.

II. Design

Currently we implement components, called stream handlers (SHs), that work at a granularity similar to the components of the JMF and do not provide an abstraction from the network. The SHs are modular media processing units that can be connected dynamically by a controlling entity, the graph manager, to form a set of modules, which process data units sequentially. The sequence of data units is called the stream, the modules are the SHs. As streaming graph we define a graph built from several SHs through which data is "flowing" while it is processed as necessary.

Our experience with the implementation of streaming applications showed us the need for a generic architecture to handle continous media streams. This became specifically clear during the development of our experimental KOM-Player platform. The platform is used for investigations on AV distribution systems. Therefore is should not only offer support for different encoding formats, transport protocols, but it must support a variety of distribution mechanisms that we investigate. Such distribution mechanisms may combine unicast and multicast distribution or may apply segmentation and reordering for efficient delivery. This led to our decision to build an environment that is based on a stream handler architecture. It was our intention to create an SH architecture that meets the following goals:

- **Easy to extend:** First of all the architecture should be a basis for developers to build their own SHs.
- **Well defined interfaces:** The interfaces for each single SH must be well defined to allow an easy interaction with exising ones.
- **Reuseability:** It should be easy to reuse already existing functionality.

A. *Concurrency*

Advanced, open middleware approaches that implement functions by concatenating functional modules into arbitrary graphs of independent components are able to attach scheduling mechanisms to arbitrary subgraphs [4]. While this approach is highly flexible, it requires either an operating system abstraction layer to allow arbitrary grouping, or information about the potential grouping capabilities of modules. For example, it is not straight-forward to support in the same thread a module that listens to a BSD socket with a module that waits for a POSIX semaphore to fire without wasting resources. Our implementation restricts the flexibility of the graph manager for combining stream handlers into processing units.

B. Stream and Streaming Graph

With our focus on delivery systems, we have not addressed issues in determining the functionality of stream handlers that may enable a graph manager to create an appropriate streaming graph. Rather, at this time we use well-known sub-sequences of SHs that are required for a specific task, such as data forwarding, writing to and playout from disk, buffering, or sequencing. The graph managers are responsible for the setup and destruction of the SHs, determine the interaction beween the individual SHs and represent the interface towards the application.

Specifically, a graph manager is required to deal with data packets from unexpected sources, and it must split a graph or merge graphs on behalf of the application. To handle operations such as user join or leave operations on multicast streams, the graph manager must be able to dynamically split and merge the streaming graph by setting up or removing SHs without disrupting the active data forwarding of a stream.

C. Stream handlers in Gleaning capable proxy-cache

In multimedia middleware research, dynamic reconfiguraiton of stream graphs is currently investigated from the aspect of the replacement of functions and of adaptation to changing resource availability [5].

Our requirements are orthogonal to these abilities and smaller in scale: in an RTP/RTSP delivery system our proxies must be able to handle gracefully within the data forwarding path unexpected new streams from the uplink side, pause and continue requests from the client side.

III. Implementation

The implementation consists of three applications that are sufficient for building an experimental streaming media distribution system: client, server and proxy-cache. All three of them exclusively use the SH architecture described in Section II to implement their streaming functionality. As a starting point the classes shown in Figure 1 were implemented.

A. Overview

The implementation of the KOM-Player platform aims at the development of a research prototype in the area of wide-area distribution systems for streaming media in the Internet. The initial code base considered mainly the distribution of CBR MPEG-1 system and MP3 streams, which were our initial target formats because they combine hardware- and OS-independent playback capability with an appropriate quality. Since these encoding formats do not support the scalability of encoding formats that can now and in the conceivable future be

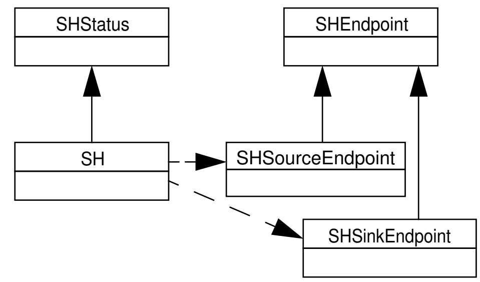

Figure 1: Parent classes

deployed in the Internet on a wide scale, more flexible encodings are considered as well in our research. More recently, we have added H.261 and VBR MPEG audio, video and system. The SH architecture will be a major basis for ongoing implementation work that is concerned with scalable encoding formats.

B. Middleware

To make SHs also usable for third party developers we decided to create a layer that provides basic classes (see Figure 1), templates and interface definitions for the creation of new SHs. Parent classes with a set of virtual functions ensure the interoperability between SHs. Certainly this can only be assured in case that newly created SHs inherit from those classes.

- **SH and SHStatus:** SH is the basis class for all SHs which must be inherited by all new SH classes. This class provides an attribute template that allows individual attributes for each SH. The SHStatus class provides functionality that allows to collect status information about a specific instance of an SH (e.g. if the SH is currently part of an active graph).
- **Endpoints:** The Endpoint classes provide standard interfaces between the SHs. Each new SH must also include a class that implements its endpoints and inherits from SHEndpoint. SHs can provide both sink and source endpoints, which must than inherits SHSinkEndpoint or SHSourceEndpoint, respectively.
- **Attributes:** Attributes of an SH are modified by the graph manager to specialize an SH before it is connected into a graph. Attributes are implemented as set/get operations on generic data types. At this time, the knowledge required for specialization is identical to the knowledge required for chosing among different SHs.
- **Reports:** It is untypical for architectures that implement uni-directional streaming of data to provide direct feedback in the opposite direction of the data path. We have decided to do this. It allows, for example, to provide RTCP feedback to an RTP packetizer without involvement of the graph manager. Each SH must implement the report interfaces (up- and downstream), and reports must be accepted in a non-blocking manner. SHs may communicate via specialized reports even if intermediate SHs can not interpret them - such reports must be forwarded.

C. Stream Handler Types

To deal with the concurrency issue, our implementation requires SHs to specify whether they implement an own clock or not, and whether they require it or not. As a result, we define three operation modes for our SHs, to be ordered appropriately by the graph manager: active, passive and through. Their combination and ordering depends on the task that a specific streaming graph should fulfil.

- **Active:** Active SHs implement their own timer. If the SH acts as a source, it will push data downstream actively (by calling a push function of the downstream SH). If it acts as a sink, it will pull data from an upstream SH actively. It may act as source and sink at the same time. The timer that is implemented by the active SH my be a local timer, or it may be implemented by observing external conditions, like user input or network packets. It is not possible to connect two active SHs directly to each other because each one tries to control synchronity. Yet, more that one active SH in a streaming graph can exist if a passive SH is inserted between those.
- **Passive:** A passive SH does not implement a clock. If it acts as a sink, an upstream SH may push data to it, if it acts as a source, a downstream SH may pull data from it. If it implements both source and sink, it must also provide buffering capacities that suite the needs of the graphs that it is likely to be included in. Such a buffering SH should define thresholds that allow it to notify the graph manager of over- and underruns of the buffer. Passive SHs can not be connected directly because no data would be exchanged between them.
- **Through:** Through SHs are meant for tasks such as on-the-fly transcoding, packet duplication, or filtering. SHs do not implement timers and should not introduce buffers beyond those necessary for their operation. They must always implement a source as well as a sink. An arbitrary number of them can be concatenated. An active SH that is located upstream will push data through this kind of SH, potentially through several more through SHs until a passive SH is encountered. The pull operation is used in the same way by an active SH located downstream. A through SH should work in both directions, but its endpoint capabilities may restrict this.

D. Client-Server Application

An example for these stream handler types' interaction is the delivery of an MPEG-1 (system stream) movie to a client. Figure 2 shows the SHs that are used in this simple scenario.

The movie is stored on the server's disk. Thus the starting point of the streaming path is an SH that reads the data from the disk. The data reader, in our case described as *File Source SH*[1], must be partially aware of the encoding format of the stored movie to schedule its read-ahead operations reasonably.

1. This is described as a source because it is the source of the streaming path.

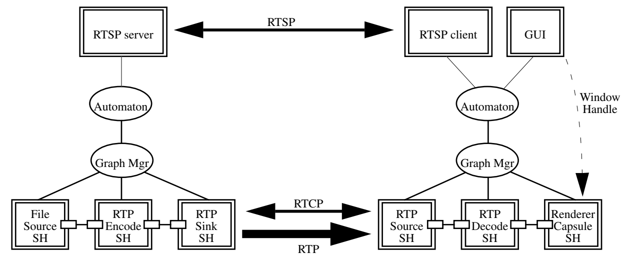

Figure 2: Client-server configuration overview

In the example data is requested from the *File Source SH* by the *Encoder SH* which is an *RTP Encoder SH* in this specific case. The encoder determines the timing in this stream. It understands the actual encoding format of the data and the transport protocol that is used for data transmission. It determines time and amount of data to *pull* from the *File Source* and *pushes* it to the *RTP Sink SH* to meet the existing constraints for data rate and delay, and to create a reasonable stream. In the case of an MPEG-1 system stream this means that the *RTP Encoder SH* requests data chunks of equal size and *pushes* those to the *RTP Sink SH*. RTCP receiver reports are interpreted by the *RTP Sink SH* and statistics are forwarded to the *RTP Encoder SH* using the report interface.

The actual streaming path is determined by the existing streaming graph which represents the layout of the streaming architecture. In Figure 2 the streaming graph consists of *Graph Manager, File Source SH, RTP Encoder SH* and *RTP Sink SH*. The *Graph Manager* is responsible for the setup and destruction of the SHs, determines the interaction beween the individual SHs and represents the interface towards the application.

To handle special tasks in caches the *Graph Manager* must be able to dynamically reconfigure the streaming graph by setting up or removing SHs.

E. Gleaning Proxy

Reconfiguration plays no role in the example of Section D but it is a basic requirement for a proxy server that implements *gleaning*. Roughly, a gleaning proxy works by delivering a movie linearly to a client via unicast, which the proxy itself receives in two pieces: a short start sequence via unicast and the remaining portion via multicast. For a detailed description of Gleaning we refer to [6].

Since one of our research topics is on caching for multimedia streams we designed and implemented a gleaning capable proxy-cache for those streams. A detailed design can be found in [7]. The proxy is not an RTSP proxy as understood in the RFC, which caches and redirects only control information [8]. Rather, it is an RTSP/RTP proxy cache that stores content in addition to handling RTSP requests. RTSP messages from different RTSP sessions are multiplexed onto one connection between an origin server and a proxy. RTSP SessionIDs are the keys to de-multiplex sessions. A proxy installs an RTSP connection to an origin server on-demand when a request for the particular origin server is received from a client. The connection is torn down when no more active RTSP sessions between the proxy and the origin server exist.

Figure 3 shows a streaming graph for the gleaning proxy

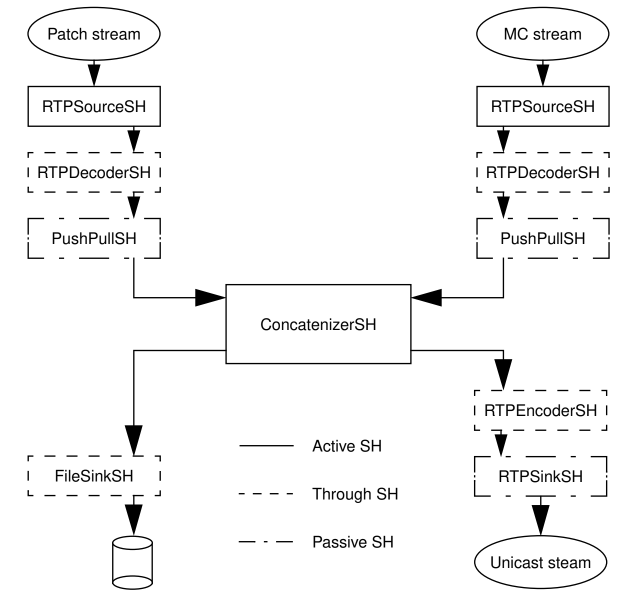

Figure 3: Proxy cache streaming graph

with a single client. In a typical case of gleaning the proxy-cache joins an already existing multicast session and requests the missing part of the movie via a unicast stream. When the missing part arrives as a unicast, this data is immediately forwarded to the client while the multicast stream is buffered cyclically and streamed to the client after the unicast stream is finished. If the proxy cache also decides to cache this movie, both streams are stored linearly on its local disc. Therefore two streaming paths on the receiving part of the cache are needed: one for the unicast stream and one for the multicast stream. This paths consist of an active *RTPSourceSH*, a passive *RTP-DecoderSH* and *PushPullSH*. The latter is needed because the *ConcatenizerSH* is active. This is the case because it determines both the order and the timing with which data is forwarded to the client or to the local disc.

On the data forwarding path to the client, an *RTPEncoderSH* can be seen in through mode. If active mode were used instead as in the previous example, the *ConcatenizerSH* and the *RTPEncoderSH* would have to be seperated by another *Push-PullSH*, and both would re-create the required timing of the RTP stream independently.

Two situations require dynamic reconfiguration of the data path:

- **join without caching:** If the proxy cache does not keep the entire movie, a second client must be served from the same multicast stream and an additional unicast stream.
- **pause with caching:** If the proxy cache keeps the entire movie, the client may decide to pause. In this case, the delivery path to the client must be suspended.

F. Ropes

Since our implementation resides in the user space, data is copied between kernel and user space at least twice in a forwarding operation. Further replication and processing is required in the streaming path. To reduce the amount of copying operations on the streaming path, we use the concept of ropes, a buffer abstraction that provides random access similar to a flat buffer, but that allows copy-by-reference combined with independent modifications of each copy by concatenation, cutting and editing operations [9]. Ropes allow the non-copying modification, removal or addition of protocol headers, and the parallel processing of interleaved channels in a streams.

IV. Conclusions

We use this SH implemenation which is based on the architecture presented in Section II in our proxy-cache prototype. We consider the SHs are an appropriate abstraction for developing streaming applications. This is supported by the fact that most of the implementation work was done by one of our students who was not involved in the SH design.

For investigate layered video and other adaptive capabilities, we will integrate an MPEG-4 encoder and -decoder into the system. This happens in conjunction with an experimental TCP-friendly protocol TFRC that indicates limits to the permitted transmission rate to the sender.

Although the applicability and extensibility of the approach has been shown, we expect a better handling when we have integrated the SH approach consequently into our client as well, and when we support a plugin architecture that allows the dynamic loading of stream handlers.

V. References

[1] L. DeCarmo. *Core Java Media Framework*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1999.

[2] T. Kaeppner. *Entwicklung verteilter Multimedia-Applikationen*. Vieweg Verlag, 1997.

[3] F. Eliassen and J. Nicol. Supporting Interoperation of Continuous Media Objects. *Theory and Practice of Object Systems: Special Issue on Distributed Object Management*, 2(2):95–117, 1996.

[4] E. Walthinsen. GStreamer - GNOME Goes Multimedia. Technical report, GUADEC 2001, April 2001.

[5] F. Kon, M. Rom·n, P. Liu, J. Mao, T. Yamane, L. C. Magalh„es, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), New York, USA, 2000*. IFIP/ACM, April 2000.

[6] C. Griwodz. *Wide-area True Video-on-Demand by a Decentralized Cache-based Distribution Infrastructure*. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany, April 2000.

[7] R. Becker. Design und Implementierung von Patching in die KOM VoD Umgebung. Studienarbeit. Fachbereich Elektrotechnik und Informationstechnik, Darmstadt University of Technology, September 2001.

[8] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326 - Real Time Streaming Protocol (RTSP). Standards Track RFC, April 1998.

[9] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An Alternative to Strings. *Software Practice and Experience*, 25(12):1315–1330, 1995.