DESIGN AND IMPLEMENTATION OF PROGRAMMABLE MEDIA GATEWAYS

Wei Tsang Ooi [*], *Robbert van Renesse and Brian Smith*

Department of Computer Science, Cornell University
Ithaca NY 14853

ABSTRACT

Treating the network as a processor that can perform computation has several benefits. Processing at strategic locations in the network may reduce bandwidth requirements. Low-powered devices that are connected to the Internet can be off-loaded as well. In this paper we present *Degas*, a programmable media gateway system. Degas allows users to upload small programs, called *deglets*, into a Degas gateway to filter, transform or mix video streams from a multicast session. We describe a declarative, event-driven programming model for writing deglets. We also discuss a simple mechanism used by gateways to optimize and execute the operations specified in the deglets. Finally, a method for selecting a suitable gateway to run deglets is outlined.

1. INTRODUCTION

In the traditional model of distributed computing, nodes on the edges of a network perform computation, and nodes inside the network move data around. Recently, researchers have been studying how computation can be moved into the network itself. This shift is motivated in part by the increasing number of low-powered devices connected to the Internet. Computationally intensive operations can be moved from these devices to nodes within the network. Performing operations on packets within the network can also improves network efficiency, (e.g. by compressing and decompressing data streams across a bottleneck link [17], or transcoding a video into a lower bandwidth within a heterogeneous network [2]). Active Networking [14] takes the idea further by making the nodes with the network programmable. Programmability allows users to extend the network with customized operations, such as application-specific retransmission scheme, or new routing protocols.

In this paper, we present the design and implementation of a programmable, application-level media gateway called *Degas*.[1] Degas allows users to "inject" user-defined programs, called *deglets*, into a gateway to perform customized transcoding, filtering and mixing of video and audio streams of a multicast session. Transcoding allows transformation of the media streams into a different format or bit rate, thus allowing heterogeneous hosts to participate in the same session over connections with different bandwidths. Filtering allows hosts to block streams from certain sources. Mixing provides processing on multiple streams. For example, a gateway can merge incoming video streams into a single video stream by creating a "picture-in-picture" or a "quad-splitter" view, or a gateway can switch between different streams in a tele-conference based on who is currently talking.

The most significant difference between Degas and previous work is the programmability that Degas provides. Instead of providing a fixed set of services, Degas allows users to upload new functionality into gateways. This simplifies deployment of new services, promoting user innovation. Also, it allows users to customize existing services.

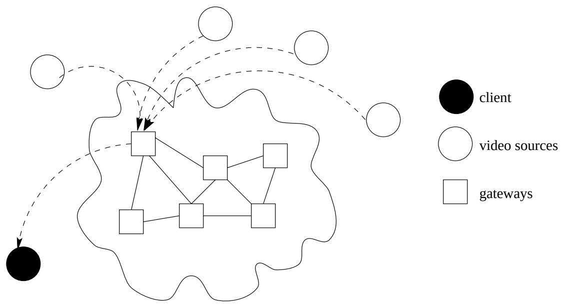


Figure 1: An example of a Degas system

Figure 1 shows an example of a Degas system. Multiple Degas gateways are distributed across the Internet. The existence of these gateways is transparent to the various senders that multicast video streams onto their respective sessions. Such transparency allows current MBone applications such as vic [7] and ivs [15] to be used with Degas without modification.

A user who is interested in receiving videos from a session through Degas runs a Degas client. The client program

[*]This research was supported by DARPA/ONR (contract N00014-95-1-0799), and grants from the National Science Foundation, Kodak, Intel, Xerox, and Microsoft.

[1]Named after French impressionist Edgar Degas.

(called degasclient), is a modified version of vic, extended with abilities to talk to the gateways and a user interface to select and use a deglet. The client first requests a service from Degas. Degas selects a gateway with enough capacity. The client then uploads its deglet into this gateway. The gateway joins the session requested by the client and runs the deglet. The processed video stream is sent to a new multicast session, which the client is listening to. A reliable control channel is also established between the client and the gateway. This control channel allows user to interact with the deglet and the gateway, such as to reconfigure deglet, send user interface events (for instance, mouse click) or migrate the deglet to another gateway. The gateway uses the same control channel to send error messages back to the client for debugging. Figure 2 shows an example output stream produced by a Degas gateway.

1.1. Research Problems

In the design and implementation of Degas, several problems arose. We briefly discuss each problem below. The first problem is related to the programming model of deglets. A deglet must be simple to specify, yet powerful enough to perform useful operations on media streams. We could allow user to write arbitrary code and submit them to the gateway for execution (as in J-Kernel [5]), but we think that this is unnecessarily since Degas is not meant to perform arbitrary computation. We should restrict the programmers to a set of API for manipulating media streams.

The second problem concerns the execution of deglets. As media processing involves large amounts of data, normally encoded in a complex format, it is crucial that the gateway executes a deglet efficiently. The optimum way of performing an operation is usually tied to the format of the input streams and output streams. Since the format of input streams may be different and can changed in the middle of a session, we must optimize the deglets differently for different streams, and re-optimize when input format changes.

The third problem is deciding where to run deglets, to optimize load balancing and use of network resources. Running a deglet at a strategic location can reduce bandwidth consumption significantly. The dynamics of the network environment complicates the problem. Gateways may be created and removed; senders and receivers may join or leave multicast sessions; and available bandwidth of a link changes from time to time. Hence, our solution for locating a gateway must be an adaptive one.

Finally, we need to ensure that the system is robust in the face of crashes and badly-behaved deglets, such as one that enters an infinite loop or allocates a huge amount of memory. Resources, including CPU, memory and network bandwidth, must be shared fairly among different deglets. Furthermore, the effect of resource controls on QoS must be minimized.

As Degas is still a work in progress, it is not our intention to solve all these problems in this paper. This paper focuses on the programming model and execution model of deglets. We briefly describe our solution for the gateway location problem and refer interested readers to [10] for details. We are still looking at a solution for the resource management problem.

1.2. Organization

The rest of this paper is organized as follows. The programming model is described in Section 2. The optimization and execution of deglets are described in Section 3. The mechanism for selecting a gateway to run a deglet is presented in Section 4. We provide some performance data in Section 5. Ongoing and future work is described in Section 6. Finally, we provide an overview of related work in Section 7 and conclude in Section 8.

2. PROGRAMMING MODEL

The main consideration in selecting a programming model for deglets is simplicity while retaining flexibility and power. We want to make deglet easy to write, so that a user can specify one in a few minutes. This consideration favors the use of scripting languages. For Degas, we chose Tcl [11]. We also chose to use a declarative model for programming deglets. A declarative model lets the user specify what to do, but not how to do it. The user should not be concern with how the deglet is going to be executed. The optimal way to perform the video operations depends heavily on the properties of the input streams, such as the encoding format and sizes. By decoupling the properties of the source media streams from the deglet specification, the same specification can be used on sources with different properties. Furthermore, the users do not have to worry about cases where sources change their transmission properties in the middle of a session. The underlying execution engine determines the best way to perform a task.

2.1. Examples

To better understand how a deglet is written, we present two examples in Figure 3 and Figure 4. We explain these two examples in detail in the rest of this section.

Figure 3 shows a simple deglet that transcodes a video stream from host `seminar.cs.cornell.edu` into a M-JPEG video stream of quality 40. A deglet is a text file that starts with a list of key-value pairs. The key `sources` specifies a list of sources we are interested in (line 1) using multiple host addresses or a regular expression such as *.cornell.edu. In this case, we are only interested in one source. `num_of_sources` indicates the maximum number

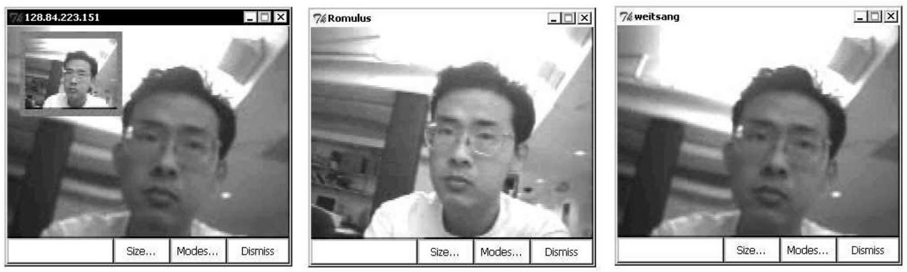


Figure 2: The left window shows the output from a deglet that creates a picture-in-picture effect. The right windows show the input streams.

of sources. `input_video_session` indicates the multicast address and port number of the input video session. `output_format` specifies the format of the processed video stream. In this example, we want to receive a 176 × 144 M-JPEG stream with quality 40.

The remainder of the deglet specifies the operation to perform when an event happens. Line 5 to 7 define a callback function to be called whenever a frame is received. The function body is defined using Tcl. We use a predefined API `frame_copy` to copy the input frame `inf` into the output frame `outf`. `frame_copy` performs the necessary scaling and transcoding operations to convert the output frames into the format specified above. The argument `src_id` identifies the source of the input stream. Since we have only one source in this case, it is not used. We show how `src_id` is used in our next example.

```
1 sources {seminar.cs.cornell.edu}
2 num_of_sources {1}
3 input_video_session {224.4.4.4/4444}
4 output_format {JPEG 40 QCIF}

5 recv_frame_callback { src_id inf outf } {
6     frame_copy $inf $outf
7 }
```

Figure 3: A simple deglet.

Our second example reads video streams from multiple sources, and outputs a ”split” video stream that consists of video from the current speaker and previous speaker. Video from other sources are filtered. For simplicity, we assumes that the number of sources is always larger than two. We explain this deglet below.

```
1  sources {*}
2  num_of_sources {*}
3  input_video_session {224.4.4.4/4444}
4  input_audio_session {224.4.4.5/4444}
5  output_format {H261}

6  init_callback { outf } {
7    set w2 [expr [frame_get_width $outf]/2]
8    set lf [frame_clip $outf 0 0   $w2
         [frame_get_height $outf]]
9    set rf [frame_clip $outf 0 $w2 $w2
         [frame_get_height $outf]]
10   set prev 0
11   set curr -1
12 }

13 talk_start_callback {src_id} {
14   if {$src_id != $curr} {
15     set prev $curr
16     set curr $src_id
17   }
18 }

19 recv_frame_callback { src_id inf outf } {
20   if {$src_id == $curr} {
21     frame_copy $inf $rf
22   } else if {$src_id == $prev} {
23     frame_copy $inf $lf
24   }
25 }

26 destroy_callback {} {
27   frame_free lf
28   frame_free rf
29 }
```

Figure 4: A more elaborate deglet example.

Line 1 - 5 specify the input and output parameters. The function `init_callback` in line 6 to 12 is called at the beginning of the deglet execution. Here, we split the output frame into the left half and the right half, denoted by variables `lf` and `rf` respectively. We also initialize the variables `curr` and `prev` that denote the source id the current speaker and the previous speaker. The function on line 13 to 18 (`talk_start_callback`) is called whenever a talk spurt is detected. The parameter `src_id` indicates the source of the talk spurt. In this function, we simply update the variables `curr` and `prev`. Note that variables set in one callback is accessible from other callbacks. In line 19 to 25, `recv_frame_callback` checks if the input frame is from source `curr` or `prev`. If it is from either of these, we copy the frame into the left half or the right half of the output frame. Finally, line 26 to 28 define the function to call when the deglet exits. We free the memory allocated for `lf` and `rf` here.

We summarize the lists of available keys, callbacks and frame operations in Table 1, 2, 3 respectively. This list is by no means complete, as we plan to add more operations to Degas. In particular, it would be interesting to add vision-related routines such as face detection and object tracking.

As illustrated in the two examples above, a deglet is a high-level, declarative style specification. They are short and simple to write. Our most complicated deglet so far, is one that creates a ”task bar” of incoming video streams, and let user maximizes or minimizes a video stream by clicking on the task bar. This is written in under 100 lines of code. Our examples also illustrate how a user can construct the output stream using ”frame” as an abstraction, without knowing what the input formats are. The Degas execution module is responsible for translating these specifications into optimized low-level code. We describe the execution of deglets next.

3. EXECUTION OF DEGLETS

The Degas execution module is responsible for parsing the deglet specification and for efficient execution of the callbacks. The execution module must recognize optimizations and translate the high-level API into appropriate low-level code. For instance, in Figure 3, if the input video stream is also in M-JPEG format, then we can employ compressed domain processing techniques to scale and copy the input frames to output frames efficiently. Furthermore, if the input video streams are already in the format requested by the user, the execution module should simply copy the streams without decoding it.

We used our low-level, high performance media processing library called *Dali* [9] as our target for the translator. The executing module is just a Tcl interpreter extended with Dali commands. Dali consists of a small set of abstractions suitable for representing commonly used video and audio formats. Dali is designed with high-performance in mind, often sacrificing ease of use for efficiency. It exposes intermediate structures of video and audio objects, such as DCT blocks, giving programmers (or in our case, the translator) the flexibility of writing highly optimized programs. Dali is also designed with predictability in mind—memory allocations and I/O operations are separated from the processing—so that the programmers have full control over memory usage and I/O. These features make Dali ideal for forming the basis of our execution module.

The optimizations and executions are carried out as follows. We defined a set of optimized versions of Dali subroutines for each high-level APIs. These high-level APIs are then bound, at run-time, to one of the subroutines based on input and output formats, dimensions and color decimation. Each call to a high-level function will cause the optimized version of the function to be executed. The high-level functions are re-bound whenever a change in input video properties is detected.

4. SELECTING GATEWAYS

Besides the key-values pairs described above, a deglet may contain a set of *preconditions*. The purpose of preconditions is to allow the user to restrict their deglets to be run on gateways that meet certain criteria. The user might impose some restrictions to improve quality of the output, or for security concerns. For example, a user might want to run his deglet on a low-load, high-capacity gateway in the same domain. The currently supported preconditions are as follows:

- `address_test`: a regular expression that matches the host addresses or IP addresses of the gateways eligible to run the deglet.
- `latency_test`: the maximum latency between the client and the gateway. This can prevent an “out-of-the-way” gateway to be assigned to the client.
- `load_test`: the maximum acceptable CPU load on a gateway.

An example of using preconditions is shown in Figure 5. This test restricts gateways to those in domain *.cornell.edu, or gateways that are within 500 ms away. Many other tests are possible in the future. For instance, the user might want to select gateways with sufficient memory, or gateways with special hardware for media processing. If the user has to pay for services on a gateway, the user may want to select gateways below a certain price.

```
precondition {
   [address_test *cornell.edu] ||
   [latency_test] < 500
}
```

Figure 5: An example of using preconditions.

When a client requests for service, the preconditions are sent along with the request. A gateway that receives a request first performs the test, and offers its service only if the test succeeds.

sources	The sources this deglet is interested in.
num_of_sources	Maximum number of sources this deglet can process.
input_video_session, input_audio_session	Specify the input video and audio session respectively.
output_format, output_size, output_fps, output_bps	Specify the format, dimension, frame rate and bit rate of the output stream.
precondition	The conditions that a gateway must satisfy before it can serve this deglet.
description	Textual description of what this deglet does.
controlling_clients	Clients that are allowed to control and modify this deglet.

Table 1: A summary of available keys in deglet specification.

init_callback(outf)	Executed when the deglet starts. `outf` is the output frame.
destroy_callback	Executed when the deglet stops.
new_source_callback(src_id, inf)	Executed when a new source is detected. `src_id` is the the source identifier. `inf` is the input frame.
del_source_callback(src_id)	Executed when a source identified by `src_id` leaves the session.
recv_frame_callback(src_id, inf, outf)	Executed when a frame from source `src_id` is received. `inf` is the received frame. `outf` is the output frame.
mouse_click_callback(x, y)	Executed when a mouse click is detected at coordinate (x,y) on the output window of the client.
input_resize_callback(src_id, inf)	Executed when input dimension of source `src_id` is changed.
talk_start_callback(src_id)	Executed when a talk spurt is detected from source `src_id`.
talk_stop_callback(src_id)	Executed when the beginning of a silence period is detected from source `src_id`.

Table 2: A summary of available callbacks in deglet specification.

frame_new w h	Return a new frame of width `w` and height `h`.
frame_copy src dest	Copy the content of frame `src` into frame `dest`, scale if necessary.
frame_clip f x y w h	Create a ”virtual” frame from frame `f`, at offset (`x`, `y`) and with dimension `w` $\times$ `h`.
frame_free f	Deallocate frame `f`.
frame_get_width f	Return the width of frame `f`.
frame_get_height f	Return the height of frame `f`.
frame_set_color f r g b	Set the color of the frame `f` to (`r`, `g`, `b`).

Table 3: A summary of available frame operations in deglet specification.

We have developed a control protocol called the Adaptive Gateway Location Protocol (AGLP) [10] for locating an appropriate gateway. AGLP optimizes network bandwidth utilization by strategically placing deglets on gateways. A deglet that transcodes to a lower bandwidth format reduces bandwidth and is best run near the source. On the other hand, if a deglet increases bandwidth consumption, it should be run close to the client. AGLP adapts to a changing environment: senders may join and leave sessions, and gateways may be added or removed. AGLP periodically evaluates the set of eligible gateways, and migrates deglets to better gateways when necessary. AGLP handles gateway and client crashes gracefully by only maintaining soft state.

5. PERFORMANCE

To better understand the overhead introduced by a Degas gateway, we ran some experiments to measure the delay caused by various components in Degas. In our experiments, we ran a Degas gateway on a Pentium II 266 MHz PC. Video streams were sent using vic from hosts connected to the gateway using an 100 MB Ethernet. Receivers, running either vic or degasclient, were located on the same LAN. We ran NTP [8] on all hosts to get a reasonably accurate measurement of end-to-end delay.

To verify that our execution model is efficient, we ran an experiment to measure the overhead introduced by our optimizer and the savings caused by the optimization. In the first experiment, the sender sent a 352×288 H261 video stream at 8 frames per second. The client requested the gateway to transcode the stream into a Motion JPEG video stream of size 176×144. We measured the time spent in the Dali interpreter for each frame received.

In the first scenario, we let the optimizer decide how to scale the frames. The optimizer detects that the output size is half the input size, and calls a specialized subroutine that shrinks the frame by half. The average time spent in scaling a frame was 2.84 ms. In the second scenario, we bypassed the optimizer, and called the optimized scaling routing ourselves. The average time spent in scaling is 2.31 ms. Finally, we turned the optimizer off, and used a general purpose scaling routine to scale the frames. The average time spent in scaling a frame increase significantly to 43.8 ms. This experiment confirmed our belief that the overhead in optimizing is small (< 1 ms), while the savings are significant (about 150%).

We also measured the total delay introduced by the decoder, encoder and the Dali interpreter when running different deglets. While these measurements were performed on specific deglets only, they give some intuition about the latency introduced by Degas's processing pipeline. A summary of our measurements is listed in Table 4. The table shows that the delay introduced by the decode-process-encode pipeline is reasonably small.

Our next two experiments measured the total end-to-end delay between the source and the receiver. This is a measurement between the time a frame is captured at the source and the time the frame is rendered at the receiver. In the first experiment, we collected the data using a degasclient. The gateway was running a deglet that shrinks the size of an incoming Motion JPEG video stream by half at 6 frames per second (deglet 2 in Table 4). For comparison, we collected the same data using vic, which received the original stream. The end-to-end delays for both experiments are shown in Figure 6. The difference between the two measurements is small, and is about the same as the total time spent in the decode-process-encode pipeline (27.5 ms). We also measured the inter-frame rendering delays in the same experiments. Figure 7(a) and Figure 7(b) show that Degas gateway introduces some jitter, but are within a tolerable level (within 20 ms).

All our performance measurements shown above are done with a single client. When the gateway serves multiple clients, the jitter increases significantly to as much as 200 ms. There is also a difference between the QoS received by the clients. The reason is that we have not implemented any resource management in Degas yet. We discuss the current implementation status and developments that we plan to do in the next section.

6. IMPLEMENTATION AND FUTURE WORK

Degas is implemented using C++ and the Mash toolkit [6]. Although still under heavy development, a preliminary prototype is available. Simple frame processing API and optimization scheme is implemented as a ”proof-of-concept”. We plan to release Degas into the MBone community in near future. Several interesting problems remain to be solved. We outline some of these problems below.

We are looking into how a client can submit multiple deglets that can be composed to perform interesting operations. A deglet that scales down two video sources and merge them into a new video streams is best separated into two stages. The first stage scales the video, and should be run near to the individual sources. The second stage combines the video, and should be run near the receiver. By using three deglets (two for scaling and one for merging), we can achieve better bandwidth efficiency than a single deglet could have achieved.

We plan to look into how Degas can control the resources of the gateways and allow deglets to be executed fairly. We want to prevent a malicious deglet from hogging a gateway. Currently, one can write a deglet that performs `frame_copy` 100 times for each frame received. Some form of scheduling has to be added so that a gateway can distribute its resources fairly. A particularly interesting is-

	Operation	Input 1	Input 2	Output	% CPU	Time
1	Shrinking	H261 352×288 at 10 fps		H261 176×144 at 10 fps	14%	9.76 ms
2	Shrinking	JPEG 320×240 at 6 fps		JPEG 160×120 at 6 fps	18%	27.5 ms
3	Picture-in-picture	H261 352×288 at 10 fps	H261 176×144 at 10 fps	H261 176×144 at 20 fps	40%	20.4 ms
4	Picture-in-picture	H261 352×288 at 10 fps	H261 176×144 at 10 fps	JPEG 176×144 at 20 fps	60%	32.4 ms

Table 4: Latencies introduced by the decode-process-encode pipeline and the CPU load incurred for different deglets.

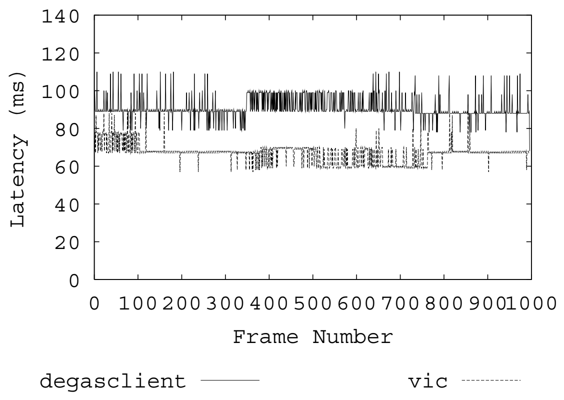
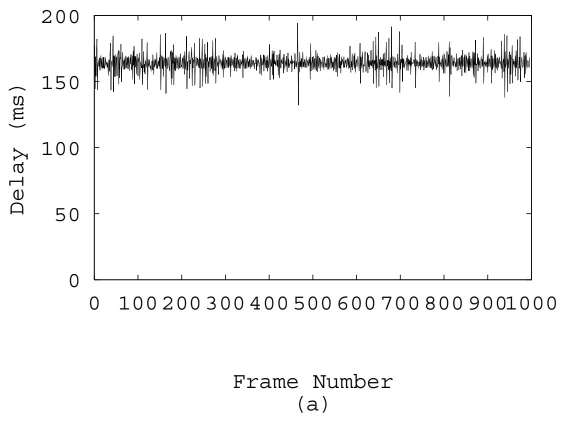


Figure 6: End-to-end Delay between the sender and the receiver.



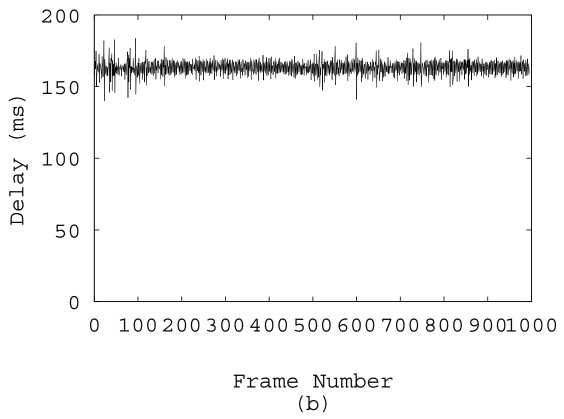


Figure 7: (a) Inter-frame rendering delay using Degas. (b) Inter-frame rendering delay without Degas.

sue is how a gateway can revoke resources from a running deglet when the gateway reallocates its resources. Revoking CPU time and bandwidth can affect the QoS received by the client. Revoking allocated memory blocks may require changing the behaviour of the deglet itself.

Security is another common concern in extensible architectures. We believe that these concerns can be easily addressed. As in Safe-Tcl [12], we can restrict the set of functions available to a deglet. This set can depend on the client that submitted the deglet. In this case, the client would have to sign the deglet, and include its digital certificate (e.g., X.509 [4]).

7. RELATED WORK

The idea of running media processing within the network was first described by Turletti and Bolot in [16] and by Pasquale et al in [13]. Turletti and Bolot suggested video gateways as a solution for solving the network heterogeneity problem. Pasquale et al proposed a filter propagation mechanism in multicast dissemination trees. By propagating filters up and down the multicast trees network efficiency may be improved. In [18], Yeadon describes a set of QoS filters that implements the idea in [13]. Unfortunately, the system is not designed to be compatible with the MBone tools, and is therefore not widely deployed. Closer to our work, MeGa [2] is an application-level media gateway that performs transcoding on RTP media streams. An advantage of Degas over previous work is that Degas allows user-defined processing on the streams, while MeGa and Yeadon's QoS filters only support a fixed set of operations.

Active Service [3] provides *clusters*, which are sets of nodes that provide certain services. User can request instantiation of an application-level service agent, such as the MeGa video gateway, on a cluster. If not available already, the agent can be uploaded. In contrast, Degas provides extensibility at a finer granularity by allowing users to extend an existing service, rather than requiring an entire new service agent to be uploaded.

8. SUMMARY

This paper describes a flexible and extensible media gateway system called Degas, which allows users to request customized processing on media streams. Degas is efficient and simple to use, while being compatible with existing popular MBone tools. Degas is also scalable in the number of gateways, and robust in the face of gateway and client crashes.

Another contribution of this paper is the Degas programming model for writing media processing specification. We use a declarative style, event-driven, scripting-based syntax to achieve simplicity, while powerful enough to specify many commonly used operations. We presented a simple model of execution, in which high-level APIs are dynamically bound to optimized low-level routines, without requiring a full-fledged compiler or optimizer.

9. REFERENCES

[1] *The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)*, San Francisco, CA, USA, November 1995. ACM Press.

[2] E. Amir, S. McCanne, and Z. Hui. An application level video gateway. In *Proceedings of the 3rd ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)* [1], pages 255–266.

[3] E. Amir, S. McCanne, and R. Katz. An Active Service framework and its application to real-time multimedia transcoding. In *Proc. of ACM SIGCOMM*, Vancouver, Canada, August 1998.

[4] C. C. I. T. T. Recommendation X.509. *The Directory-Authentication Framework*, 1988.

[5] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in java. In *Proc. of the 1998 USENIX Annual Technical Conf*, pages 259–270, New Orleans, LA, 1998.

[6] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T. L. Tung, D. Wu, and B Smith. Toward a common infrastucture for multimedia-networking middleware. In *Proceedings of 7th. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'97)*, St. Louis, Missouri, May 1997.

[7] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *Proceedings of the 3rd ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)* [1].

[8] D. L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992.

[9] W. T. Ooi and B. Smith. Dali : A multimedia software library. In *Proceedings of Multimedia Computing and Networking*, San Jose, CA, January 1998.

[10] W. T. Ooi and R. van Rennese. An adaptive protocol for locating media gateways. Technical Report submitted to ACMMM2000, Department of Computer Science, Cornell University, 2000.

[11] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[12] J. K. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl security model. Technical Report TR-97-60, Sun Microsystems Laboratories, March 1997.

[13] J. C. Pasquale, G. C. Polyzos, E. W. Anderson, and V. P. Kompella. Filter propagation in dissemination trees: Trading off bandwidth and processing in continuous media networks. *Lecture Notes in Computer Science*, 846:259–269, 1994.

[14] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, January 1997.

[15] T. Turletti. The INRIA videoconferencing system. *ConneXions - The Interoperability Report Journal*, 8(10):20–24, October 1994.

[16] T. Turletti and J. Bolot. Issues with multicast video distribution in heterogeneous packet networks. In *Packet Video Workshop*, pages F3.1–3.4, Portland, Oregon, September 1994.

[17] M. Yarvis, A. A. Wang, A. Rudenko, P. Reiher, , and G. J. Popek. Conductor: Distributed adaptation for complex networks. Technical Report CSD-TR-990042, University of California, Los Angeles, Los Angeles, CA, August 1999.

[18] N. Yeadon, A. Mauthe, D. Hutchison, and F. Garcia. QoS filters: Addressing the heterogeneity gap. *Lecture Notes in Computer Science*, 1045:227–244, 1996.