DESIGN AND IMPLEMENTATION OF DISTRIBUTED PROGRAMMABLE MEDIA GATEWAYS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Wei Tsang Ooi

August 2001

DESIGN AND IMPLEMENTATION OF DISTRIBUTED PROGRAMMABLE MEDIA GATEWAYS

Wei Tsang Ooi, Ph.D.

Cornell University 2001

Multicasting multimedia streams over the Internet is problematic due to network and host heterogeneity. One of the proposed solutions is to place media gateways inside the network to adopt the media streams for different links and hosts.

In this dissertation, we investigate several novel extensions to the existing model of media gateways. First, we make the gateways user extensible by allowing injection of Tcl code that specifies transformations on the media streams. Second, we improve efficiency in bandwidth consumption by adaptively running the transformations at strategic locations on the Internet. We further reduce bandwidth consumption and improve throughput by decomposing a transformation into multiple sub-transformations for execution on different gateways, forming a data flow pipeline. To realize these extensions, we have designed, implemented, and simulated several components of the media gateways.

We designed and implemented Dali, a low-level software library for building high-performance, predictable, highly extensible and computationally intensive multimedia applications. We designed Dali based on a set of design principles that is different from previous media processing libraries. We sacrificed ease of use for performance. We exposed the underlying structure of the media data (such as

DCT blocks and motion vectors), forced explicit resource control, and promoted sharing of memory. As a result, programs written in Dali are fast, more predictable and highly re-configurable. Dali serves as a case study in API design for high performance media processing libraries.

We built a prototype of a programmable, application-level media gateway, called Degas. Using an event-driven, descriptive programming model, users can write simple programs, called deglets, that can be uploaded into the gateway to perform operations on input video frames. We perform per-operation optimization by mapping a high-level API to low-level Dali code. Our prototype serves as a framework where many research issues about the design of programmable media gateways can be explored.

We designed, simulated, and implemented AGLP, which is an application-level protocol for choosing strategically located Degas gateways on a wide-area network to run deglets. AGLP minimizes bandwidth consumption, and is able to adapt to a changing network environment by migrating deglets. We use the announce/listen paradigm and multicast damping to achieve robustness and scalability. Our experimental results show that AGLP is able to scale up to a large number of gateways on the network.

Finally, we investigated a mechanism for distributing a deglet onto multiple gateways. By modelling a deglet as a tree of operations, we use a linear time algorithm to decompose a deglet into multiple sub-deglets. We also extended AGLP to locate helper gateways to run these sub-deglets. These sub-deglets form a pipeline where video streams can flow through. Our experiment results shows that we are able to improve thoughput and quality over a bottleneck gateway, without compromising the scalability of AGLP.

Biographical Sketch

Wei Tsang Ooi was born in October, 1971 in Alor Setar in the northern state of Kedah, Malaysia. After he finished his secondary education at Keat Hwa Secondary School in 1991, he moved to Singapore to pursue his undergraduate degree in the Department of Information System and Computer Science, at the National University of Singapore. He graduated with a First Class Honors Degree in 1996 and joined his alma mater as a senior tutor. In the same year, he took a study leave and went to Cornell University in Ithaca, New York to pursue a Ph.D. degree in Computer Science.

To my parents

Acknowledgements

First of all, I would like to thank my advisors Dr. Robbert van Renesse and Dr. Brian C. Smith. I could not have completed my Ph.D. degree without them.

Robbert took me under his wings during the third year of my graduate study, and has given me precious advice on research, paper writing, and personal matters. He continuously supports me in my decisions, and gives me freedom in choosing my own research path. I am also grateful to him for patiently reading through my papers and dissertation, and for giving me constructive criticism to improve my research and writings.

Brian introduced me to the field of multimedia, and gave me a direction for my research. He never stopped throwing ideas at me. Brian has taught me many valuable lessons during the early years of my graduate study. He taught me many skills required for research, including writing papers, reviewing papers, preparing slides and making presentations. He motivated me to improve my English accent and social skills as well.

Both Robbert and Brian have made me a better researcher, and a better person in general.

I also would like to thank Prof. Zygmunt Haas, who has kindly agreed to serve on my committee as my minor advisor. I am thankful to Prof. Emin Gun Sirer for serving as a proxy for Brian during my B-exam. I would like to thank Dr. Srinivasan Keshav, Dr. Praveen Seshadri, Dr. Thorsten von Eicken, Prof. Ramin Zabih, Prof. Lloyd N. Trefethen, Prof. Kenneth Birman, and Prof. Eva Tardos for their help and valuable advice they have given me during my five years as a graduate student at Cornell University.

Cindy Robinson has made my life much easier. She has helped me with my travel arrangements and other administrative matters. Cindy has also encouraged me to improve myself as a professional, by teasing me and telling me about the dos and don'ts.

I would like to acknowledge the friendship and mentorship of my fellow graduate students Soam Acharya, Sugata Mukhopadhyay and Tibor Janosi. They have helped me greatly when I first joined the Zeno Multimedia Research Group. I am also grateful to the Dali team, who worked very hard in building and releasing Dali. I deeply appreciate the quiet late night companions of Chris Hawblitzel, Deyu Hu and Chi-Chao Chang during my overnight stays in the system lab. The lab has been very different without them.

The support from the Mash group at the University of California, Berkeley was invaluable in getting me acquainted with the Mash toolkit. I would like to especially thank Ketan Mayer-Patel for his help during my one-week visit to Berkeley and valuable e-mail exchanges about Dali and my work.

I have had the great fortune to be a student of Dr. Leong Hon Wai and Dr. Tay Yong Chiang during my undergraduate study at the National University of Singapore. They have inspired me to do interesting research in Computer Science and played a great role in my decision to come to Cornell.

The National University of Singapore has kindly allowed me to take a 5 years study leave from my post to pursue my graduate study. I am deeply grateful to them for this.

Life as a Ph.D. student can be difficult sometimes. I am blessed to have Wei Hsin Wang by my side. Wei Hsin constantly supports me during my studies and encourages me when my moral is low. She continues to charm me with her extreme cuteness. Her love and caring motivates me to achieve my best.

Finally, I am forever indebted to my parents and my family for their teaching, understanding, and support. They are why I am the person I am today.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet usage has exploded in recent years. The amount of traffic on the Internet approximately doubles every year [16]. Data sent over the Internet is no longer restricted to text and images. Audio and video traffic are increasingly common. For example, a study at University of Washington shows that at least 18% to 24% of Web related traffic into the university is multimedia content [82]. 20% to 30% of traffic measured on University of Wisconsin-Madison campus is due to Napster, a music swapping application [61]. The popularity of digital media will only increase in the future. Digital video and audio will be accessible not only from desktop machines, but will be available on devices like PDAs [41, 70], watches [45], cell phones [37] and even refrigerators [81].

This increase in multimedia popularity is driven by a few factors. First, advances in compression technology, such as MPEG [28], make it feasible to store and transport digital video and audio. Second, advances in hardware technology, such as faster processors and integration of MMX instructions [17] into processors, improve the speed of media compression and decompression. Third, higher bandwidth into the home using DSL and cable modem allows significant improvement in the quality of video and audio received via the Internet. Finally, new applications are utilizing video and audio to deliver contents over the network. These applications include distance learning, entertainment broadcast, tele-conferencing, and surveillance.

A common requirement of these applications is many-to-many communication over the Internet. The original one-to-one communication model of the Internet is unsuitable for these applications. To address this weakness, IP Multicast [20] was introduced in the late 80s as an extension to the Internet Protocol (IP). Multicast allows data packets to be sent to multiple destinations efficiently, by replicating data packets in routers only when necessary. Hence, even with many receivers, the sender only needs to send one copy of the packet. Unfortunately, this led to new problems due to the heterogeneous nature of the Internet.

1.1 Network and Host Heterogeneity

The Internet is an inherently heterogeneous environment, with many different types of connections, and hosts with different capabilities. The types of connections range from high speed campus/corporate networks, to medium speed DSL/ISDN/Cable modems, to slow dial-up connections. End hosts on the Internet include super computers, powerful desktops, low powered appliances and personal digital assistants (PDA). Due to the decentralized nature of the Internet, the Internet will likely remain heterogeneous in the future.

Network and host heterogeneity causes problems for delivering multicast media streaming over the Internet. How can we transmit a single stream that suits different receivers with different bandwidth? A common solution that many web sites are adopting is to simulcast multiple video streams in different formats and bit rates for different classes of receivers. This solution is not scalable and wastes resources. Another common solution is just to send a single stream. However, using the lowest maximum bit rate that suits every user often sacrifices quality. Sending a higher bandwidth stream improves quality, but isolates users with slow connections. Furthermore, this addresses only the issue of bandwidth heterogeneity and not host heterogeneity.

As bandwidth and CPU speed increase, this heterogeneity gap widens. Meanwhile, the demand for streaming media applications increases as well. This presses for a solution to the heterogeneity problem.

1.2 Programmable Media Gateways

This dissertation describes an approach to solving the Internet heterogeneity problem. Our solution places programmable media gateways inside the network. These media gateways can transform media streams from a multicast session into suitable formats for different receivers. Possible transformations include transcoding, filtering and mixing. 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.

While the primary goal of media processing in the network is to adapt media streams to heterogeneous end hosts, it can also avoid waste of bandwidth. The rationale is that we do not want to send unnecessary data that has no use to the receivers. For example, if a receiver is a PDA that can only display gray scale video at 5 frames per second, it is wasteful to send a color video stream at the full frame rate.

Our work extends previous work in media gateways in three ways. Our media gateway provides user extensibility, network efficiency and distributed processing. These extensions form the basis of our contributions in this dissertation.

1.3 Contributions

We now present a brief overview of our contributions in this dissertation.

1.3.1 A High-Performance Media Processing Library

In order to build a media gateway that performs processing-intensive operations on media streams, we need a high-performance media processing software library. Although many software libraries exist, we found that many of these emphasize ease of use, and often sacrifice performance. Furthermore, getting these software libraries to work with each other is a time-consuming task. To address this issue, we built a new software library, called *Dali*, that focuses on performance rather than ease of use. Dali provides a small set of abstractions to represent common multimedia data types (images, video, audio), and a set of low-level operations to operate on these abstractions. Throughout the design of Dali, we follow a few design principles that are different from previous media processing libraries. By exposing structural elements in a media format, explicit resource control and sharing of memory, we make programs written with Dali fast, has better predictability and highly configurable. We describe Dali and its design principles in detail in Chapter 3.

Dali is a contribution that is useful independent of building our media gateway. It represents a case study in designing an API for a high performance media processing library. It also benefits the research community as a common toolkit for media processing. So far, Dali has been used in many multimedia research projects, including Microsoft, Inktomi, the University of California Berkeley, the University of North Carolina, and Cornell University.

1.3.2 User Extensible Media Gateway

We implemented a prototype of our media gateway called *Degas*. Degas allows a user to submit a small program (known as a *deglet*) to specify the media transformation to be performed on the gateways, thus providing an extensible environment for user customization. Since the evaluation of video and audio quality is often subjective and user dependent, customization allows users to receive the media streams the way they prefer. This extensibility also simplifies deployment of new transformations, and promotes user innovation. Our implementation of Degas provides a working prototype of a programmable media gateway, where further research issues can be explored.

To provide a simple and flexible programming model for users to compose their deglets, we designed an event-based, descriptive programming language for writing deglets. Our goal is to make writing deglets an easy task, where one can compose a deglet in a few minutes. Degas and the deglet's programming model is described in Chapter 4.

1.3.3 Adaptive Gateway Location Protocol

We envision that multiple Degas gateways will be deployed across the network. Hence, when a user requests a transformation to be performed, we have to choose one of the gateways to service a client. There are several factors that can affect the decision. Our solution chooses a gateway for the purpose of reducing network bandwidth consumption.

We designed, simulated, and implemented an application-level protocol, called Adaptive Gateway Location Protocol, or AGLP, which locates media gateways in the network, and chooses a strategically located gateway to service a client. AGLP can adapt to changes in the environment, including birth and death of senders and gateways, by migrating services from one gateway to another. Our results show that AGLP is scalable to a large number of gateways.

We believe that AGLP can be easily adapted to other applications that involve transforming large amounts of data inside the network, for example, distributed database and data fusion. We describe AGLP in Chapter 5.

1.3.4 Distributing Media Processing Over Multiple Gateways

Our last contribution is a mechanism to distribute a service over multiple gateways. Services on multiple gateways can be composed to perform a high-level transformation on a media stream. By flowing through multiple gateways, multiple operations can be performed on a media stream before it reaches the receivers.

This in effect creates a data-flow pipeline on the streams. By composing services on multiple gateways, we can better distribute the load among the gateways. This can lead to higher throughput. Running an operation over multiple gateways can also reduce bandwidth consumption further. Lastly, output from an intermediate service can be shared if it is needed by different users requesting different services.

We extend the AGLP protocol to locate gateways for running composable services. We call the new protocol AGLP++. We also present a linear-time algorithm for decomposing a computation into sub-computations for execution on multiple gateways. We describe this contribution in Chapter 6.

Together, Dali, Degas, AGLP and AGLP++ contribute towards the realization of a distributed, programmable media gateway architecture.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 presents some background information, and describes recent work related to our contributions. Chapter 3 describes the Dali software library and the design principles behind it. We present Degas and the deglet programming model for our gateway in Chapter 4. AGLP is described in Chapter 5. Our decomposition algorithm and AGLP++ are described in Chapter 6. We conclude in Chapter 7.

Chapter 2

Background and Related Work

In this chapter, we present the technological backgrounds needed to understand the rest of this dissertation. This is followed by an overview of various research work related to our own.

2.1 Multimedia Multicast

IP Multicast

Traditional Internet communication is one-to-one. This communication model is inefficient for applications that require one-to-many, or many-to-many communication. To address this weakness, researchers have developed an extension to IP called IP multicast [20]. IP multicast allows hosts to send a single message to multiple receivers. Routers duplicate the data packets only when necessary, hence reducing unnecessary packets inside the network. The paths the packets flow through form a tree, with the source as root and the receivers as leaves. This tree is known as the *multicast tree* (See Figure 2.1).

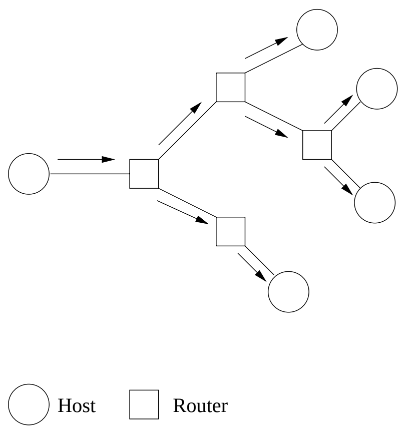


Figure 2.1: Multicast

IP multicast provides a group model for communication. Senders and receivers communicate through a multicast group, identified by a *multicast address*. The multicast group provides a level of indirection and decouples the senders from the receivers. Hence the senders need not know the identities of the receivers. They simply send data to a multicast group. To receive data, a host simply *joins* the multicast group.

In IP multicast, a receiver will receive data from all sources sending to a group it has joined. There is no provision for preventing traffic from unwanted sources. This can lead to waste of bandwidth, and possibly denial of service attacks. To address this issue, Source-Specific Multicast [35] was proposed as an extension to IP multicasting. Under this new model, a host may join a multicast group, but only choose to receive data from a particular subset of sources sending to the group. Unlike IP Multicast, a receiver needs to know the identity of the senders. At this time of writing, Source-Specific Multicast is still a draft and yet to become an IETF standard.

Due to its efficiency in one-to-many and many-to-many communications, IP multicast has been used in applications such as teleconferencing or distance learning to disseminate video streams to multiple receivers. The video streams are often sent on a UDP channel on top of IP multicast. However, UDP does not provide enough mechanism to support time-sensitive continuous media data. The needed mechanisms are added in an application-level protocol, RTP, which we describe next.

RTP

The Real-time Transport Protocol (RTP) [67] is an application-level protocol that is designed to meet the needs of transporting continuous multimedia data. It extends the underlying transport protocol with the following features:

- payload type identification - to identify the type and format of the data carried by a packet. This lets the receiving application know how the data should be decoded.
- sequence numbering - for packet loss and packet re-ordering detection.
- timestamping - packets are timestamped with the time when the data carried is sampled. This timestamp, known as "RTP timestamp", is offset by a random amount for stronger encryption. Hence it does not correspond to wallclock time. However, this can still be used to determine the play-out time of the packet since only a relative time difference is needed.

RTP normally runs on top of UDP and IP multicast to provide best-effort, many-to-many communication for networked media applications.

The Real-Time Control Protocol (RTCP) is a companion control protocol for RTP. It is used to send control information that provides reception quality feedback, cross-media synchronization, and sender identification.

Reception feedback is done via RTCP packets known as Sender Report (SR) and Receiver Report (RR). SR contains information such as the number of bytes and the number of packets sent so far. The receiver uses SR packets to calculate the packet loss rate. This loss rate, along with other statistics, such as inter-arrival jitter is sent using an RR packet back to the sender. Senders can use the RR packet to identify problems in the network and adjust their output stream rates.

Cross-media synchronization can be done by embedding an NTP [52] timestamp along with RTP timestamp in SR packets. This allows applications to determine a mapping between both timestamps. The corresponding sampled time between two streams can then be derived.

Finally, RTCP packets can contain several fields that describe the source, such as the name, e-mail, phone number, geographical location, and a textual description of the sender.

RTP was first conceptualized in an audio conferencing tool called *vat*. Vat is one of the earliest multicast media applications. In the next section, we give a brief overview of an early multicast-enabled network called MBone and the historic set of applications running on it.

MBone and MBone Tools

MBone [22], or Multicast Backbone, is a set of interconnected hosts with a multicast routing capability. It was originally a virtual network overlayed on top of the physical Internet to provide multicast capability. MBone links groups of hosts that run a multicast routing daemon, `mrouted`, and tunnels through portions of the network that do not have multicast support using unicast. In recent years, increasing numbers of routers with native multicast support were deployed into the network. Hence, MBone now interlinks nodes with native multicast capability and nodes running `mrouted`.

The MBone served as a testbed for multicast applications in the early 90's. Some of the earliest MBone applications include video conferencing tools (vic [50], nv [26]), audio conferencing tools (vat [39], nevot [66], rat [34]), shared white boards (wb [40], mb [76]), and session directory tools (sdr [31]). These applications provided a platform where many research issues in multicast multimedia applications were studied. In 1998, most of these tools were re-implemented. Their common functions were abstracted, and were integrated into a toolkit called Mash. We describe Mash, and the scripting language it is based on, Tcl, next.

2.2 Software Tools

Tcl

Tcl, or Tool Command Language, [55] is an interpreted, scripting language. Tcl treats all data types as a string, hence is able to provide a simple syntax. Tcl can be extended with user commands, and can be embedded in applications. Due to these desirable features, Tcl was chosen to be the base language for Mash.

Mash

Mash [49] is a toolkit for building remote collaborations and streaming media applications. It was built by merging the functionalities and design ideas from the MBone tools, the VuSystem from M.I.T [44], and the Continuous Media Toolkit (CMT) from University of California, Berkeley [71]. Mash is built using C++ and an object-oriented version of Tcl, called OTcl [80]. It consists of a set of reusable and flexible components that can be composed to perform a function. Since Mash abstracts away many low-level details, such as communication with video capturing devices and the encoding/decoding process, it allows applications to be written simply by gluing together these high-level components. For example, the following two lines of Mash code create a window that displays received media streams from session 224.8.15.90 at port 4000.

```
set agent [new VideoAgent "224.8.15.90/4000" ... ]
$agent attach [new VideoUI ... ]
```

Classes in Mash can be defined in both OTcl and C++. This *split object* model allows a programmer to write low-level code in C++ for performance, and glue objects together using OTcl for convenience.

2.3 Techniques for Multicast Communication

Now, we look at various techniques used in designing multicast media applications. Most RTP-based multicast media applications leverage the "host group" model from IP multicast. A host that wishes to receive data from a multicast group simply joins the group. It can leave the group to stop receiving data. Other members already in the group need not be notified about the joining and leaving of a member. Similarly, the joining host need not know about other members already in the group. This loosely-coupled communication model is also known as the *light-weight session* architecture [38].

2.3.1 Announce/Listen Paradigm

A common technique used to design communication protocols for a light-weight session is the *announce/listen* paradigm. An announce/listen-based protocol has the following properties :

- Multiple parties communicate over a shared multicast channel.
- Each party maintains a state.
- Each party periodically sends update messages to the shared channel.
- States are updated, or *refreshed*, by the periodic messages. States that are not refreshed get deleted after a timeout period. Such aging states are also known as *soft states* [15].

The announce/listen-based protocol is appropriate when it is sufficient to achieve "eventual consistency" among the states of the participants. The use of periodic messages and soft-states allows failures to be detected, and recovery to be performed during normal operation of the protocol. No recovery phase is needed. States pertained to a failed participant will eventually be deleted, and a recovered participant can be brought up-to-date by periodic messages from other participants. Hence, robustness is "built-in" to the protocol. The announce/listen paradigm has been used in the design of many protocols. For example, the Internet Group Management Protocol (IGMP) [23] uses announce/listen to maintain group membership in a router. The Session Announcement Protocol [33] uses an announce/listen-based mechanism to advertise multicast sessions to session directories [31]. The Soft State Archive Control (SSAC) Protocol [65] and Active

Service Control Protocol (ASCP) [5] are announce/listen-based protocols for accessing media archiving servers and Active Service clusters respectively.

2.3.2 Multicast Damping

A common problem in designing multicast communication protocols is *feedback implosion*. This happens when multiple receivers respond to the sender at once, causing the sender to be overwhelmed by replies. This can be triggered by a particular type of messages from the sender, such as a query, or by a lost message.

A widely used technique to avoid implosion is *multicast damping*. It allows redundant replies that cannot affect the state of the system to be suppressed. Multicast damping works as follows. Instead of sending a reply immediately, a receiver first sets a timer, and waits for a certain amount of time. The receiver replies after the timer expired. However, if it receives a reply from another receiver that makes its own reply redundant, the receiver suppresses its reply and nothing is sent.

Multicast damping is commonly used to improve scalability. IGMP [23] uses multicast damping to avoid multiple replies when a router queries its subnet for group membership. Scalable Reliable Multicast (SRM) [24] uses multicast damping to consolidate NACK messages from the receivers when a lost message is detected. The Active Service Control Protocol (ASCP) [5] uses it to avoid multiple service launches when a service is requested.

2.4 Related Work

In this section, we review recent research results in areas related to our work. We begin by looking at existing tools for building processing-intensive multimedia applications.

2.4.1 Existing Multimedia Software Libraries

There are many libraries for processing multimedia data. Most of these libraries work on a specific data format. Among the popular libraries, NETPBM [19] provides processing routines for the PGM, PPM and PBM image formats; IJG JPEG [30] provides API for JPEG compressed images; gd [10] works on GIF and PNG formats, and ooMPEG [36] provides a high-level API to retrieve frames from MPEG video streams.

Some use a scripting language for processing multimedia data (VideoScheme [46], Isis [2], Rivl [73]). These scripting languages are typically high-level, weakly typed, interpreted languages that support the composition of components and rapid prototyping. They provide high-level commands for manipulation of multimedia data. Isis and VideoScheme do not address performance issues in processing. Rivl addresses performance issues by using optimization techniques such as lazy evaluation and memory management. However, the optimizations that Rivl can perform are limited because Rivl combines the interpreter, optimizer and execution engine into one single monolithic system. This combined function makes Rivl difficult to extend and debug.

PPE [62] is a multimedia toolkit that uses composable components to construct multimedia software. For example, PPE includes components such as a Huffman decoder, a zigzag decoder, and an IDCT decoder that can be pipelined to build a JPEG decoder. PPE is designed to provide configurable modules that can adapt themselves to the heterogeneous environment, but not for more general multimedia processing. PPE is meant for building decoders whose components can be easily replaced. For instance, a fast, inaccurate IDCT can be used instead of a slower, more accurate IDCT when CPU power is limited.

2.4.2 Media Processing in the Network

Transforming World Wide Web images and text using HTTP proxies to adapt to heterogeneous network and hosts has been studied by several projects, for instance, TranSend [25], Mowser [8] and Digestor [9]. These proxies typically perform distillation on web objects, by re-authoring the HTML contents and reducing image resolutions, to reduce transmission time and allow display of web contents on a small screen.

The idea of running video processing within the network was first described by Turletti and Bolot in [78] and by Pasquale et al. in [57]. Turletti and Bolot suggest video gateways as a solution for solving the network heterogeneity problem. Pasquale et al. propose a filter propagation mechanism in multicast dissemination trees. A filter is a transformer of one or more input streams into an output stream. By relocating filters to appropiate points in the multicast trees, network efficiency can be improved.

In [85], Yeadon describes a set of QoS filters that implement the idea by Pasquale et al. They implemented several filters, including transcoding, low pass filters and frame dropping, which can sit inside switches or gateways and transform video streams as they flow through. These filters are dynamically instantiated by a *filter daemon* when requested by users. They also provide a graphical user interface for end users to reconfigure the filters.

MeGa [4] is an application-level media gateway that performs transcoding on RTP media streams. It is implemented as a *service agent* in the Active Service framework [5]. Active Service provides *clusters*, which are sets of nodes that provide services. A user can request instantiation of an application-level service agent on the cluster. If not available already, the agent can be uploaded. Users can implement their own service agents in Mash and run them on the clusters.

The Active Service framework provides customized processing in the network at application-level. There is also much interest on running user code in the network at the network-level [3, 74, 86]. An Active Network provides programmability at the packet level, and allows users to customize operations inside routers and switches, such as routing, caching, retransmissions, and so on. Several projects have used Active Networking to perform multimedia QoS adaptation and resource reservation in routers [75, 63, 64].

2.4.3 Locating Gateway Services

Locating services in the network is a common problem and many protocols exist. For example, DHCP [21] uses a centralized server at a known location to provide information about the location of local DNS servers. DHCP is intended for local area networks only – DHCP does not scale, and its centralized design makes it vulnerable to crashes.

SLP [79] uses another approach, where each server (or *service agent* in SLP) periodically announces availability of services to a well-known multicast channel. A client (or *user agent* in SLP) can discover services available by listening to the multicast channel. An optional directory agent acts as an intermediate agent between users and servers by caching service advertisements on behalf of the users. A client can send a request for service either to the directory agent, or multicast the request to all servers. Other protocols similar to SLP exist, for instance, SDS [18] and Jini [51]. All of these protocols are aimed at discovery of a wide range of services, such as printers, fax machines and music repositories.

Other discovery protocols not only discover services, but try to locate services at a strategic location. MeGaDiP (Media Gateway Discovery Protocol) [84] is one such protocol. MeGaDiP is used to locate media gateways for transcoding media streams from a sender to a receiver. It uses centralized directory agents called *dealers* to maintain a list of available gateways. An end host contacts a local dealer to find a gateway. If no gateway is available, the dealer forwards the request to another dealer along the end-to-end path. Lists of dealers along the path are obtained using `traceroute` and modified DNS lookup.

MeGaDiP saves network traffic by locating a media gateway near the sender or the receiver based on the property of the transcoding operation to be performed. If the operation reduces bandwidth of the media stream, then discovery is performed by the sender. Otherwise, it is performed by the receiver. By sending the request from the appropiate end host, MeGaDiP can improve bandwidth consumption. Furthermore, locating a gateway along the end-to-end path reduces increments in propagation delay caused by the gateway.

2.4.4 Distributed Media Processing

Video processing contains a high level of parallelism. This parallelism is exploited and studied in Parallel Software-Only Video Processing (PSVP) [48, 47]. They employed multiple hosts in a network-of-workstation environment to exploit temporal parallelism and spatial parallelism in video processing. In temporal parallelism, a host demultiplexes a video stream and sends different frames to different hosts for processing. The results are then sent to another host, which merges them into the resulting stream. For example, the demultiplexing host can send odd numbered frames to one host and even numbered frames to another. In spatial parallelism, different regions of a video frame are sent to different hosts for processing. A third type of parallelism, functional parallelism, allows multiple hosts to perform different operations on a video stream as it flows through the hosts. This parallelism, however, is not exploited in PSVP. The three types of parallelism are illustrated in Figure 2.2.

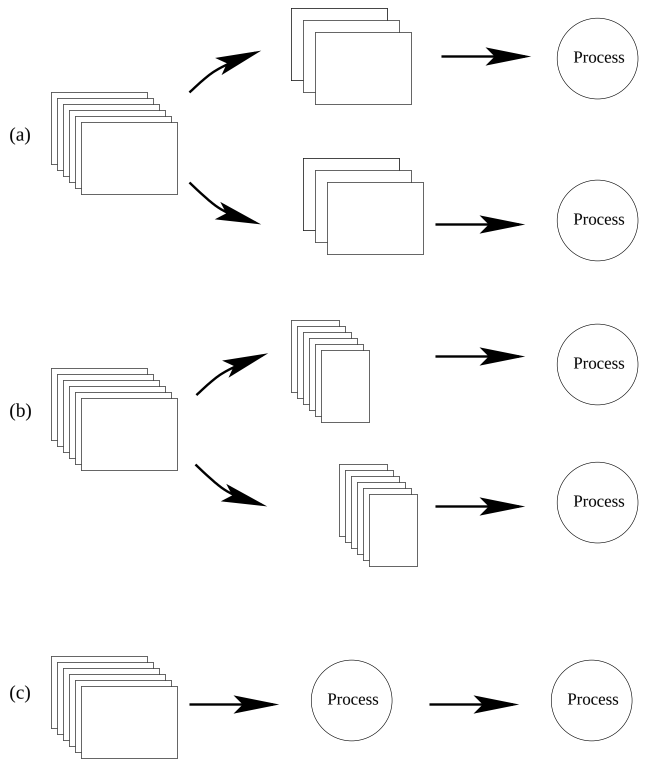


Figure 2.2: Parallelism in video processing. (a) temporal parallelism, (b) spatial parallelism, and (c) functional parallelism.

Distributed media processing is also used in the Active Service [5] framework in the form of *service composition.* In Active Service, a media gateway running as a service agent can act as a client to request another media gateway to be instantiated as its server. This can continue recursively and result in a chain of media gateways, forming a "higher-level" service to serve the client.

Several wide-area network services allow composition of their services. CANS [27] (Composable, Adaptive Network Services) composes its services in response to a client's request using a centralized plan manager. The composition of services is transparent to the client. The plan manager constructs a data path through the services, using a heuristic to maximize the minimum bandwidth available along the path. Ninja [29] uses a different heuristic to compose its services. Their automatic path creation facility first maps a user request to a data path with a minimum number of operators, and then assigns these operators onto the least loaded servers on the network.

2.5 Conclusion

We described the basic networking technology we use in our work. The Degas media gateway is designed to process RTP-based, multicast video streams and is backward compatible with the MBone tools. We use the announce/listen-based protocol and multicast damping to achieve robustness and scalability. Degas and AGLP are built using the Mash toolkit to leverage existing building blocks for constructing a networked media application.

Next, we gave an overview of related research. Our contributions draw upon the strength of this previous work, and address some of its weaknesses.

Chapter 3

The Dali Multimedia Software Library

Our research in building a distributed programmable media gateway is aligned with recent trends in multimedia research. Traditionally, the multimedia research community has focused much of its efforts on the compression, transport, storage, and display of multimedia data. These technologies are fundamentally important for applications such as video conferencing and video-on-demand, and results from the research community have made their way into many commercial products. For example, JPEG [60] and MPEG [28] are now ubiquitous standards for image and audio/video compression.

Although many research problems remain in these areas, the research community has begun to examine the systems problems that arise in multimedia data processing, such as content-based retrieval and understanding [59, 69], video production [83], and transcoding for heterogeneity and bandwidth adaptation [1, 4]. Our work falls into the last category.

The lack of a high-performance toolkit that the community can use to build processing-intensive multimedia applications is hindering this research. Currently, researchers have several options (none of which are adequate). They can develop code from scratch, but the complex nature of common multimedia encoding schemes (e.g., MPEG) makes this approach impractical. For example, several person-years of work went into writing the Berkeley MPEG player (`mpeg_play`) [58].

A more commonly used option is to modify an existing code base to add the desired functionality. For example, many researchers have "hacked up" `mpeg_play` to test their ideas. However, this approach requires understanding thousands of lines of code and usually results in complex, unmanageable systems that are difficult to debug, maintain, and reuse.

A third option is to use standard libraries, such as ooMPEG [36] or the Independent JPEG Group (IJG) software [30]. Such libraries provide a high-level API that hides many of the details of the underlying compression scheme. However, since programmers cannot penetrate the "black-box" of the API, they can only exploit limited optimizations. For example, to extract a gray scale image from an MPEG frame, the programmer must convert the RGB image returned by the ooMPEG library into a gray scale image. A much more efficient strategy is to extract the gray scale image directly from the MPEG frame data, since it is stored in a YUV color space. The ooMPEG abstractions do not support this optimization. Another problem is that these libraries usually provide functions for specific multimedia formats, making interoperability between the libraries difficult. For example, it is difficult to transcode an MPEG I frame into a JPEG image, although both of them are DCT coded.

These concerns have led us to develop Dali, a library for constructing processing-intensive multimedia software. Dali consists of a set of simple, inter-operable, high-performance primitives and abstractions that can be composed to create higher level operations and data types. Dali lies between the high-level APIs provided by common libraries and low-level C code. It exposes some low level operations and data structures but provides a higher level of abstraction than C, making it possible to compactly write high-performance, processing-intensive multimedia software. Dali's mechanisms include:

- Resource control. Programmers have full control over memory utilization and I/O. With few exceptions, Dali routines do not implicitly allocate memory or perform I/O - such functions are always explicitly requested by the programmer. This feature gives programmers tight control over performance-critical resources, an essential feature for writing applications with predictable performance. Dali also gives programmers mechanisms to optimize their programs using techniques such as data copy avoidance and structuring their programs for good cache behavior.

- "Thin" primitives. Dali breaks complex functions into simple functions that can be layered. This feature promotes code reuse and allows optimizations that would otherwise be difficult to exploit. For example, to decode a JPEG image, Dali provides three primitives: (1) a function to decode the bit stream into three `SCImage`s, one for each color component (a `SCImage` is an image where every "pixel" is a structure containing DCT coefficients), (2) a function to convert each `SCImage` into a `ByteImage` (an uncompressed image whose pixels are integers in the range 0..255), and (3) a function to convert from YUV color space to RGB color space. Exposing this structure has several advantages. First, it promotes code reuse. For instance, the inverse DCT and color-space conversion functions are shared by the JPEG and MPEG routines. Second, it allows optimizations that would be difficult to exploit otherwise. For example, compressed domain processing techniques [4, 69, 59, 72] can be implemented on `SCImages`.

- Exposing Structure: Dali provides functions to parse compressed bit streams, such as MPEG, JPEG, and GIF. These bit streams consist of a sequence of structural elements. For example, an MPEG-1 video bit stream consists of a sequence header followed by one or more group-of-pictures (GOPs – Figure 3.5). Each GOP is a GOP header followed by one or more pictures. Each picture is a picture header followed by encoded picture data. While other libraries hide these structures from programmers, Dali exposes them. Dal provides five functions for each structural element: find, parse, encode, skip, and dump. The functions operate on data in a memory buffer (call a `BitStream`). *Find* locates that element in the `BitStream`, *parse* reads the element into an associated data structure, *encode* writes a data structure into the `BitStream`, *skip* moves the `BitStream` cursor past that structural element, and *dump* copies the element from one `BitStream` to another. These routines allow a programmer to operate on a bit stream at a high level, but to perform operations that are impossible with conventional libraries. For example, writing a routine that counts the number of I frames in an MPEG sequence is trivial: one simply finds each picture header, parses it, and increments a counter if it is an I frame (indicated by the `type` field in the picture header structure). Similarly, writing a program to demultiplex MPEG Systems streams or analyze the structure of an MPEG sequence is very easy. Similar considerations hold for other formats, such as GIF and JPEG.

The challenge of Dali was to design a library of functions that (1) allowed us to write code whose performance was competitive with hand-tuned C code, (2) allowed us to perform almost any optimization we could think of without breaking open the abstractions, and (3) could be composed in interesting, unforeseen ways. We believe that we have achieved these goals. For example, a Dali program that decodes an MPEG-1 video into a series of RGB images is about 150 lines long, runs about 10% faster than `mpeg_play`, and can be easily modified for new environments.

The contributions of this research are two-fold. First, we believe that Dali provides a fairly complete set of operations that will be useful to the research community for building processing-intensive multimedia applications. Second, this research is a case study in designing high-performance multimedia APIs. It provides a model for what APIs operating systems should provide to programmers.

The rest of this chapter is organized around these contributions. To show how Dali is used, we describe Dali in Section 3.1 through three illustrative examples. Section 3.2 describes the design principles of Dali. Finally, the implementation and its performance are briefly discussed in Section 3.3.

3.1 Dali By Example

This section is intended to give the reader a feel for programs written using Dali. We first outline the major abstractions defined by Dali and then present three examples of programs written with Dali that illustrate its use and power.

3.1.1 Abstractions

To understand Dali, it is helpful to understand the data types Dali provides. The basic abstractions in Dali are:

- `ByteImage` - a 2D array of values in the range 0..255.
- `BitImage` - a 2D array of 0/1 values.
- `SCImage` - an image where each ”pixel” is a structure that represents the run-length-encoded DCT blocks found in many block-based compression schemes, such as MPEG and JPEG.
- `VectorImage` - an image where each ”pixel” is a structure that represents the motion-vector found in MPEG or H.261.
- `AudioBuffer` - an abstraction to represent audio data (mono or stereo, 8-bit or 16-bit).
- `ImageMap` - represents a look-up table that can be applied to one `ByteImage` to produce another `ByteImage`.
- `AudioMap` - a look-up table for `AudioBuffer`.
- `BitStream/BitParser` - a `BitStream` is a buffer for encoded data. A `BitParser` provides a cursor into the `BitStream` and functions for reading/writing bits from/to the `BitStream`.
- `Kernel` - 2D array of integers, used for convolution.
- `BitStreamFilter` - a scatter/gather list that can be used to select a subset of a `BitStream`.

These abstractions can be used to represent common multimedia data objects. For example,

- A gray-scale image can be represented using a `ByteImage`.
- A monochrome image can be represented using a `BitImage`.
- An irregularly shaped region can be represented using a `BitImage`.
- An RGB image can be represented using three `ByteImage`s, all of the same size.
- A YUV image in 4:2:0 format can be represented using three `ByteImage`s. The `ByteImage` that represents the Y plane is twice the width and height of the `ByteImage`s that represent the U and V planes.
- The DCT blocks in a JPEG image, an MPEG I-frame, or the error terms in an MPEG P- and B-frame can be represented using three `SCImage`s, one for each of the Y, U and V planes of the image in the DCT domain.
- The motion vectors in MPEG P- and B-frame can be represented with a `VectorImage`.
- A GIF Image can be represented using three `ImageMap`s, one for each color map, and one `ByteImage` for the color-mapped pixel data.
- 8 or 16-bit PCM, (μ-law or A-law) audio data (mono or stereo) can be represented using an `AudioBuffer`.

Dali also has abstractions to store encoding-specific structures. For example, an `MpegPicHdr` stores the information parsed from a picture header in an MPEG-1 video bit stream. The header abstractions in the current implementation are listed in Table 3.1.

3.1.2 Examples

Although the set of abstractions defined in Dali is fairly small (9 general purpose and 13 header abstractions), the set of operators that manipulate these abstractions is not. Dali currently contains about 500 operators divided into 12 packages. The rationale for defining so many operators is discussed in Section 3.2. It is neither practical nor productive to describe all the operators here. Instead, we present three examples that illustrate the use of the Dali abstractions and give you a feel for programs written using Dali. The first example shows how to use Dali to manipulate images, the second shows how to use Dali for MPEG decoding, and the last shows how to use a Dali `BitStreamFilter` to demultiplex an MPEG Systems stream.

Image Primitives

The first example uses Dali to perform a picture-in-picture operation (Figure 3.2). Before explaining this example, we must describe the `ByteImage` abstraction in detail. A `ByteImage` consists of a header and a body. The header stores information such as the width and height of the `ByteImage` and a pointer to the body. The body is a block of memory that contains the image data. A `ByteImage` can be either physical or virtual. The body of a physical `ByteImage` is contiguous in memory, whereas a virtual `ByteImage` borrows its body from part of another `ByteImage` (called its parent). In other words, a virtual `ByteImage` provides a form

Table 3.1: Header abstractions in Dali.

`PnmHdr`	NETPBM image header
`WavHdr`	WAVE audio header
`GifSeqHdr`	GIF file sequence header
`GifImgHdr`	GIF file image header
`JpegHdr`	JPEG image header
`JpegScanHdr`	JPEG scan header
`MpegAudioHdr`	MPEG-1 audio (layer 1, 2, 3) header
`MpegSeqHdr`	MPEG-1 video sequence header
`MpegGopHdr`	MPEG-1 video group-of-picture header
`MpegPicHdr`	MPEG-1 video picture header
`MpegSysHdr`	MPEG-1 system stream system header
`MpegPckHdr`	MPEG-1 system stream pack header
`MpegPktHdr`	MPEG-1 system stream packet header

of shared memory - changing the body of a virtual `ByteImage` implicitly changes the body of its parent (see Figure 3.1).

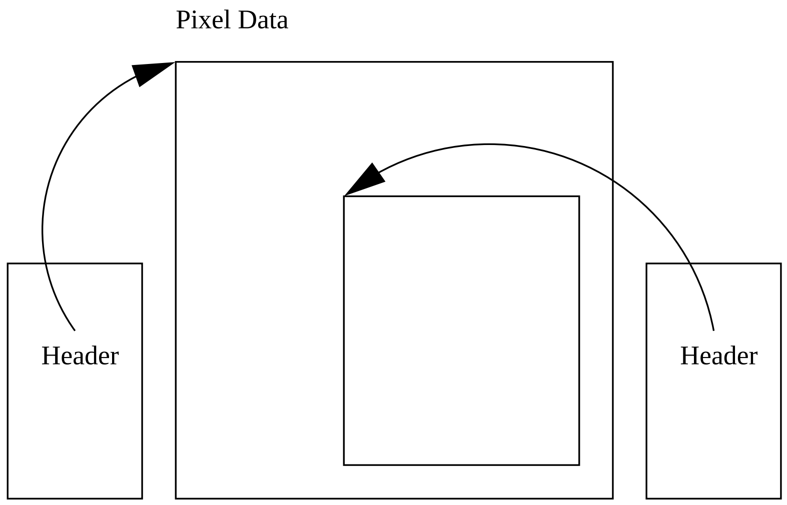


Figure 3.1: Physical (left) and virtual (right) `ByteImage`s.

A new physical `ByteImage` is allocated using `ByteNew`(w,h). A virtual `ByteImage` is created using `ByteClip`(b, x, y, w, h). The rectangular area whose size is $w \times h$ and has its top left corner at (x, y) is shared between the virtual `ByteImage` and the physical `ByteImage` b. The virtual/physical distinction applies to all image types in Dali. For example, a virtual `SCImage` can be created to decode a subset of a JPEG image.

We now show how to use Dali to create a "picture in picture" (PIP) effect on an image (Figure 3.2). We choose this example because it is simple, yet involves basic operators that illustrate the principles of Dali.

The steps to create the PIP effect can be briefly stated as follows: given an input image, (1) shrink the image by half, (2) draw a white box slightly larger than the scaled image on the original image, and (3) paste the shrunk image into the white box.

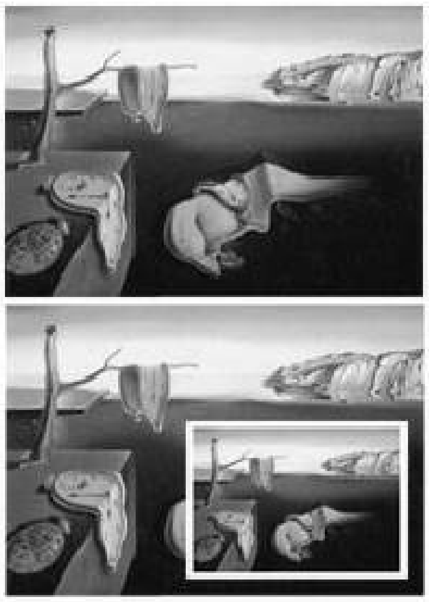


Figure 3.2: Input (top) and output (bottom) of the function PIP.

Figure 3.3 shows a Dali function that performs the PIP operation. The function takes in three arguments: `image`, the input image; `borderWidth`, the width of the border around the inner image in the output, and `margin`, the offset of the inner image from the bottom right edge of the outer image. (See Figure 3.4).

Line 5 to line 6 of the function query the width and height of the input image. Line 7 to line 10 calculate the position and dimension of the inner picture. Line 13 creates a new physical `ByteImage`, `temp`, which is half the size of the original image. Line 14 shrinks the input image into `temp`. Line 15 creates a virtual `ByteImage` slightly larger than the inner picture, and line 18 sets the value of the virtual `ByteImage` to 255, achieving the effect of drawing a white box. Line 19 de-allocates this virtual image. Line 20 creates another virtual `ByteImage`, corresponding to the inner picture. Line 21 copies the scaled image into the inner picture using `ByteCopy`. Finally, line 22 and 23 free the memory allocated for the `ByteImage`s.

```
void PIP(image, borderWidth, margin)
  ByteImage *image;
  int borderWidth, margin;
{
  int w  = ByteGetWidth(image);
  int h  = ByteGetHeight(image);
  int destW = w/2;
  int destH = h/2;
  int destX = w - destW - margin;
  int destY = h - destH - margin;
  ByteImage *dest;
  ByteImage *temp;

  temp = ByteNew(destW,destH);
  ByteShrink2x2(image, temp);

  dest = ByteClip(image,
    destX-borderWidth, destY-borderWidth,
    destW+2*borderWidth,destH+2*borderWidth);
  ByteSet(dest, 255);
  ByteFree(dest);

  dest = ByteClip(image, destX, destY, destW, destH);
  ByteCopy(temp, dest);
  ByteFree(dest);
  ByteFree(temp);
}
```

Figure 3.3: PIP function written using Dali.

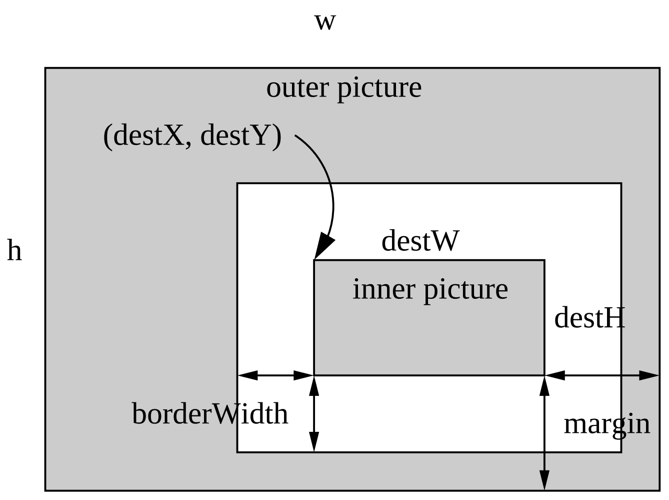


Figure 3.4: Variables used in function PIP.

This example shows how images are manipulated in Dali through a series of simple, thin operations. It also illustrates several design principles of Dali, namely (1) sharing of memory (through virtual images), (2) explicit memory control (through `ByteClip`, `ByteNew` and `ByteFree`), and (3) specialized operators (`ByteShrink2x2`). These design principles will be discussed in greater detail in Section 3.2.

MPEG and `BitStreams` Primitives

Our next example illustrates how to process MPEG video streams using Dali. Our example program decodes the I-frames in an MPEG video stream into a series of RGB images. Before discussing the example, we briefly review the format of MPEG video streams and the relevant Dali abstractions and functions.

To parse an MPEG video stream, the encoded video data is first read into a `BitStream`. A `BitStream` is an abstraction for input/output operations - that is, it is a buffer. To read and write from the `BitStream`, we use a `BitParser`. A

`BitParser` provides functions to read and write data to and from the `BitStream`, plus a cursor into the `BitStream`.

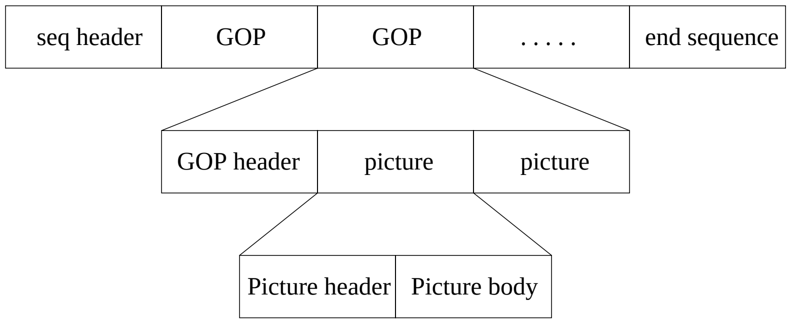


Figure 3.5: Format of an MPEG-1 video stream.

An MPEG video stream consists of a sequence header, followed by a sequence of GOPs (group-of-pictures), followed by an end of sequence marker (Figure 3.5). Each GOP consists of a GOP header followed by a sequence of pictures. Each picture consists of a picture header, followed by the compressed data required to reconstruct the picture. Sequence headers contain information such as the width and height of the video, the frame rate, the aspect ratio, and so on. The GOP header contains the time code for the GOP. The picture header contains information necessary for decoding the picture, most notably the type of the picture (I, P, B). Dali provides an abstraction for each of these structural elements (see Table 3.1), and five primitives for each structural element: find, skip, dump, parse, and encode. For example, the `MpegPicHdrFind` function advances the cursor to the next picture header, and `MpegSeqHdrParse` decodes the sequence header into a structure.

Given this background, we can describe the Dali program shown in Figure 3.6, which decodes the I-frames in an MPEG video into RGB images. Lines 1 through 4 allocate the data structures needed for decoding. Line 8 associates `inbp` to `inbs`.

```
// filename is the name of the MPEG file to parse
BitStream *inbs = BitStreamMmapReadNew (filename);
BitParser  *inbp = BitParserNew ();
MpegSeqHdr *seqhdr = MpegSeqHdrNew ();
MpegPicHdr *pichdr = MpegPicHdrNew ();
int w, h, vbvsize, status;
ScImage *scy, *scu, *scv;
ByteImage *y, *u, *v, *r, *g, *b;

BitParserWrap(inbp, inbs);

MpegSeqHdrFind(inbp);
MpegSeqHdrParse(inbp, seqhdr);

w = MpegSeqHdrGetWidth(seqhdr);
h = MpegSeqHdrGetHeight(seqhdr);
vbvsize = MpegSeqHdrGetVbvSize(seqhdr);

r = ByteNew(w, h);
g = ByteNew(w, h);
b = ByteNew(w, h);
y = ByteNew(w, h);
u = ByteNew((w+1)/2, (h+1)/2);
v = ByteNew((w+1)/2, (h+1)/2);
scy = ScNew((w+15)/16, (h+15)/16);
scu = ScNew((w+31)/32, (h+31)/32);
scv = ScNew((w+31)/32, (h+31)/32);

while (1) {
  status = MpegPicHdrFind (inbp);
  if (status == DVM_MPEG_NOT_FOUND) break;
  MpegPicHdrParse (inbp, pichdr);
  if (pichdr->type == I_FRAME) {
    MpegPicIParse (inbp,scy,scu,scv);
    ScToByte (scy, y);
    ScToByte (scu, u);
    ScToByte (scv, v);
    YuvToRgb420 (y, u, v, r, g, b);
  }
}
```

Figure 3.6: Dali code to decode I-frames from an MPEG video to RGB format.

The cursor of `inbp` will be pointing to the first byte of the buffer in `inbs`, which is a memory-mapped version of the file. Lines 9-10 move `inbp` to the beginning of a sequence header and parse the sequence header into `seqhdr`.

We extract vital information such as width, height and the minimum data that must be present to decode a picture (`vbvsize`) from the sequence header in lines 11-13. Lines 14 through 22 allocate the `ByteImage`s and `SCImage`s we need for decoding the I-frames. The variables `scy`, `scu`, and `scv` store compressed (DCT domain) picture data, `y`, `u`, and `v` store the decoded picture in YUV color space, and `r`, `g`, and `b` store the decoded picture in RGB color space.

The main loop in the decoding program (lines 23-34) starts by advancing the `BitParser` cursor to the next MPEG picture header (line 24). If the picture header is not found, we exit the loop (line 25). Otherwise, we parse the picture header (line 26) and check its type (line 27). If it is an I-frame, we parse it into three `SCImage`s, (line 28), convert the `SCImage`s to `ByteImage`s (lines 29-31), and convert the `ByteImage`s into RGB color space (line 32).

Breaking down complex decoding operations like MPEG decoding into ”thin” primitives makes Dali code highly configurable. For example, by removing lines 30 to 32, we get a program that decodes MPEG I-frame into gray scale images. By replacing line 29 to 32 with JPEG encoding primitives, we get an efficient MPEG I-frame to JPEG transcoder. Similarly, we can just as easily write a Motion-JPEG to MPEG I-frame transcoder.

BitStreamFilters

Our final example illustrates how we can filter out a subset of a `BitStream` for processing. `BitStreamFilter`s were designed to simplify the processing of bit streams with interleaved data (e.g., AVI, QuickTime, or MPEG Systems streams). `BitStreamFilter`s are similar to scatter/gather vectors - they specify an ordered subset of a larger set of data.

A common use of filtering is processing MPEG Systems streams, which consists of interleaved audio or video (A/V) streams (Figure 3.7). In MPEG, each A/V stream is assigned an unique id. Audio streams have ids in the range 0..31; video streams ids are in the range 32..47. The A/V streams are divided up into small (approximately 2 kilobytes) chunks, called packets. Each packet has a header that contains the id of the stream, the length of the packet, and other information (e.g., a time code).

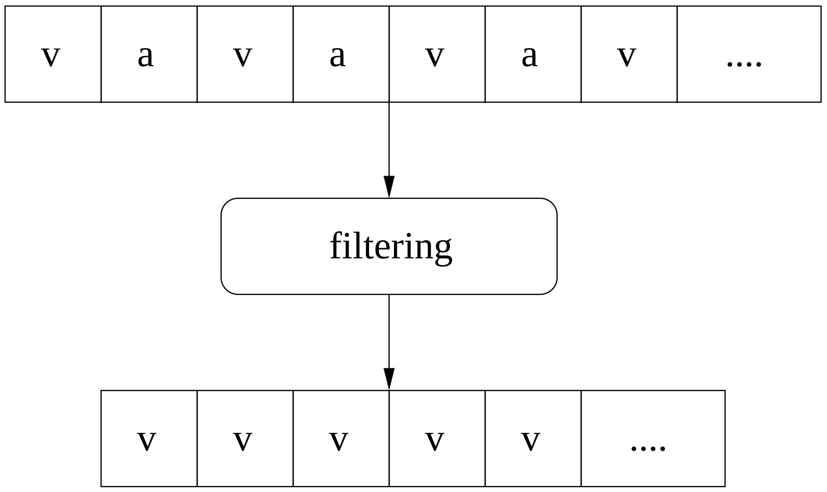


Figure 3.7: `BitStreamFilter`.

In this example, we build a `BitStreamFilter` that can be used to copy the packets of the first video stream (id = 32) from a system stream stored in one `BitStream` to another. Once copied, we can use the Dali MPEG video processing primitives on the video-only `BitStream`. The Dali code for building this filter is shown in Figure 3.8.

```
#define SIZE (128*1024)
int len, offset, start = 0;
MpegPktHdr *hdr = MpegPktHdrNew();
BitStream *bs = BitStreamNew (SIZE);
BitParser *bp = BitParserNew ();
BitStreamFilter * filter = BitStreamFilterNew();

BitParserAttach (bp, bs);
BitStreamFileRead (bs, file);

offset = MpegPktHdrFind (bp);
while (!eof(file) && !EndOfBitstream(bp)) {
  MpegPktHdrParse (bp, hdr);
  if (hdr->id == 32) {
    len = hdr->len;
    BitStreamFilterAdd(filter, offset, len);
    start += UpdateIfUnderflow (bp,bs,file,SIZE/2);
    offset = start + MpegPktHdrFind(bp);
  }
}
```

Figure 3.8: Filtering an MPEG Systems stream by copying the first video stream to another BitStream for processing.

Lines 2 through 8 allocate and initialize various structures needed by this program. The variable `offset` stores the byte offset of a packet in the bit stream, relative to the start of the stream. Line 9 advances the cursor to the beginning of the first packet header and updates `offset`. The main loop (lines 10-18) parses the packet header (line 11) and, if the packet belongs to the first video stream, its offset and length are added to `filter` (line 14). `EndOfBitstream` is a macro that checks the position of the bit stream cursor against the length of the data buffer.

Once the `BitStreamFilter` is constructed, it can be saved to disk, or used as a parameter to functions such as `BitStreamFileFilter`, which reads the subset of a file specified by the filter, or `BitstreamDumpUsingFilter`, which copies the data subset specified by a filter from one `BitStream` to another.

This example illustrates how Dali can be used to demultiplex interleaved data. The technique is easily extended to other formats, such as QuickTime, AVI, MPEG-2 and MPEG-4. Although this mechanism uses data copies, the cost of copying is offset by the performance gain when processing the filtered data. Another option (one we initially tried) is to integrate the filter mechanism directly into the bit-at-a-time parsing functions provided by the `BitParser`. Although this design avoids unnecessary data copies, we found the overhead of checking if the cursor was at a filter segment boundary on each function call too high to make this design practical. A better option would be to provide hardware support for scatter/gather vectors [14].

These three examples illustrate Dali programs. Interested readers may consult the Dali web site at `http://www.cs.cornell.edu/dali` for more details and examples.

3.2 Design Principles of Dali

One of the contributions of this research is that it provides a case study in the design of high-performance software libraries for processing multimedia data. Many of the design decisions we made differ from other libraries because Dali emphasizes performance over ease of use. This goal put us at an unusual point in the design space. In this section, we highlight the principles that emerged during the design of Dali.

Three themes emerge from these principles. The first theme is predictable performance. We designed Dali to allow programmers to easily predict the performance of their code. We believe it is important that programmers have a simple, well-defined cost model for the functions provided by a library. Predictable performance is important for writing high-performance code because it simplifies the analysis required when making design decisions between alternative implementations of a program. It is also important for writing programs that are well-behaved in real-time environments.

The cost of functions in existing libraries can be difficult to predict. This unpredictability has several sources. Often, functions will perform hidden, expensive operations, such as I/O or memory allocation. How much these operations cost, and when they occur, is hidden behind the abstractions provided by the API. For instance, many video-decoding libraries provide a function to ”get the next frame.” But the execution time for this function can be vastly different when decoding MPEG video, depending on the frame type (I, P, or B). Another source of unpredictability is that the execution time of a function can be very non-linear, depending on the value of the parameters. For example, scaling an image down by a factor of 2 can be significantly faster than scaling an image down by a factor of 1.9 if interpolation is used.

The second, closely related theme in the design of Dali is resource control. We wanted to give programmers better control over the machine's resources (at the language level, not the OS level). Predictable performance gives programmers control over their use of the CPU, but memory and I/O are very important resources in multimedia applications. Dali provides several mechanisms for giving the programmer tight control over memory allocation and I/O execution, and for reducing or eliminating unnecessary memory allocation.

The final theme is replaceability and extensibility. We wanted Dali to be usable in many applications, not just the ones we envision. For example, Dali would be useful in building a multimedia database. Since most database management systems perform their own I/O, we separated the Dali I/O functions from the computation functions. Throughout the design, we tried to make Dali extensible and pieces of it replaceable.

In summary, the design problem we faced was providing an API that was coarse enough to provide a useful level of abstraction to the programmer, yet fine enough to give the programmer tight control over their code. The following sections describe the mechanisms we use to solve this problem in detail.

3.2.1 I/O Separation

Few Dali primitives perform I/O. The only ones that do are special I/O primitives that load/store `BitStream` data. All other Dali primitives use `BitStream` as their data source.

This separation has three advantages. First, it makes the I/O method used transparent to Dali primitives. Other libraries use integrated processing and I/O. A library that integrates file I/O with its processing is difficult to use in a network environment, since the I/O behavior of networks is different from that of files. Second, the separation of I/O also allows control of when I/O is performed. We can build a multi-threaded implementation of Dali that will allow us to use a double buffering scheme to read and process data concurrently. Third, by isolating the I/O calls, the performance of the remaining functions becomes more predictable.

3.2.2 Sharing of Memory

Dali provides two mechanisms for sharing memory between abstractions. These mechanisms are called clipping and casting. In clipping, one object ”borrows” memory from another object of the same type. An example usage of clipping can be seen in Figure 3.3. Clipping functions are cheap (they only allocate an image header structure), and are provided for all Dali image and audio data types. Clipping is useful for avoiding unnecessary copying or processing of data. For example, if we only want to decode part of the gray-scale image in an MPEG I-frame, we could create a clipped `SCImage` that contains a subset of DCT blocks from the decoded I-frame and then perform the IDCT on that clipped image. The advantage of this strategy is that it avoids performing the IDCT on encoded data that we will not use.

While clipping is the sharing of memory between objects of the same type, casting refers to the sharing of memory between objects of different types. Casting avoids unnecessary copying of data. Casting is often used in I/O, since all I/O must be done through a `BitStream`. To avoid copying data, a section of the

`BitStream` buffer can be shared with another object. For instance, we can read a PGM image file into `BitStream`, parse the headers, and cast the remaining data into a `ByteImage`.

3.2.3 Explicit Memory Allocation

In Dali, the programmer allocates and frees all non-trivial memory resources using new and free primitives (e.g., `ByteImageNew` and `ByteImageFree`). Functions never allocate temporary memory - if such memory is required to complete an operation (scratch space, for example), the programmer must allocate it and pass it to the routine as a parameter. Explicit memory allocation allows the programmer to reduce or eliminate paging, and make the performance of the application more predictable.

To illustrate these points, consider the `ByteCopy` function, which copies from one `ByteImage` to another. One potential problem is that the two `ByteImage`s might overlap (e.g., if they share memory, via clipping). One way to implement `ByteCopy` is shown on the left side of Figure 3.9. This implementation allocates a temporary buffer, copies the source into the temporary buffer, copies the temporary buffers into the destination, and frees the temporary buffer. In contrast, the Dali `ByteCopy` operation assumes that the source and destination do not overlap, so it simply copies the source into the destination. The programmer must determine if the source and destination overlap, and if so allocate a temporary `ByteImage` and two `ByteCopy` calls (Figure 3.9, right).

A third possible implementation is to only allocate a temporary buffer if the source and destination overlap. This implementation has the drawback that its

```
ByteCopy(src, dest) {                ByteCopy(src, dest) {
      temp = malloc ();                  memcpy src to dest;
      memcpy src to temp;            }
      memcpy temp to dest;
      free (temp);                   temp = ByteNew ();
}                                    ByteCopy(src, temp);
                                     ByteCopy(temp, dest);
ByteCopy(src, dest);                 ByteFree (temp);
```

Figure 3.9: Possible implementations of `ByteCopy`.

performance would be difficult to predict. If the source and destination overlap, the function could take 2-3 times longer to complete than if they do not.

3.2.4 Specialization

Many Dali primitives implement special cases of a more general operation. The special cases can be combined to achieve the same functionality of the general operation, and have a simple, fast implementation whose performance is predictable. `ByteCopy` is one such primitive - only the special case of non-overlapping images is implemented.

Another example is image scaling (shrinking or expanding the image). Instead of providing one primitive that scales an image by an arbitrary factor, Dali provides five primitives to shrink an image (`Shrink4x4`, `Shrink2x2`, `Shrink2x1`, `Shrink1x2`, and `ShrinkBilinear`) and five others to expand an image. Each primitive is highly optimized and performs a specific task. For example, `Shrink2x2` is a specialized function that shrinks the image by a factor of 2 in each dimension. It is implemented by repeatedly adding 4 pixel values together and shifting the result, an extremely fast operation. Similar implementations are provided for `Shrink4x4`, `Shrink2x1`, and `Shrink1x2`. In contrast, the function `ShrinkBilinear` shrinks an image by a factor between 0.5 and 2 using bilinear interpolation. Although arbitrary scaling can be achieved by composing these primitives, splitting them into specialized operations makes the performance predictable, exposes the cost more clearly to the programmer, and allows us to produce very fast implementations.

3.2.5 Generalization

The drawback to specialization is that it can lead to an explosion in the number of functions in the API. Sometimes, however, we can combine several primitives without sacrificing performance, which significantly reduces the number of primitives in the API. We call this principle generalization.

A good example of generalization is found in the primitives that process `AudioBuffer`s. `AudioBuffer`s store mono or stereo audio data. Stereo samples from the left and right channels are interleaved in memory (Figure 3.10).

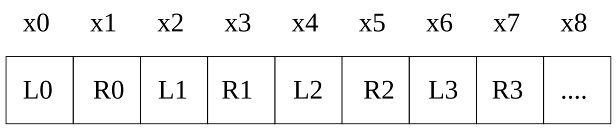


Figure 3.10: `AudioBuffer`.

Suppose you were implementing an operation that raises the volume on one channel (i.e., a balance control). One possible design is to provide one primitive that processes the left channel and another that processes the right channel (Figure 3.11(a)). However, we can combine the two without sacrificing performance by modifying the initialization of the looping variable (1 for right, 0 for left). This implementation is shown in Figure 3.11(b).

```
(a)
process_left(x)
for (i = 0; i < n; i+= 2) {
        process x[i]
}
process_right(x)
for (i = 1; i < n; i+= 2) {
        process x[i]
}

(b)
process(x, offset)
for (i = offset; i < n; i+= 2) {
        process x[i]
}
```

Figure 3.11: Generalization of an `AudioBuffer` primitives.

In general, if specialization gives better performance, it should be used. Otherwise, generalization should be used to reduce the number of functions in the API.

3.2.6 Exposing Structures

Most libraries try to hide details of encoding algorithms from the programmer, providing a simple, high-level API. In contrast, Dali exposes the structure of compressed data in two ways.

First, Dali exposes intermediate structures in the decoding process. For example, instead of decoding an MPEG frame directly into RGB format, Dali breaks the process into three steps: bit stream decoding (including Huffman decoding and dequantization), frame reconstruction (motion compensation and IDCT), and color space conversion. For example, the `MpegPicParseP` function parses a P frame from a `BitStream` and writes the results into three `SCImage`s and one `VectorImage`. A second primitive reconstructs pixel data from `SCImage` and `VectorImage` data, and a third converts between color spaces. The important point is that Dali exposes the intermediate data structures, which allows the programmer to exploit optimizations that are normally impossible. For example, to decode gray scale data, one simply skips the frame reconstruction step on the U/V planes. Furthermore, compressed domain processing techniques can be applied on the `SCImage` or `VectorImage` structures.

Dali also exposes the structure of the underlying bit stream. As described in the introduction and Section 3.1.2, Dali provides operations to find structural elements in compressed bit streams. This feature allows programmers to exploit knowledge about the underlying bit stream structure for better performance. For example, a program that searches for an event in an MPEG video stream might cull the data set by examining only the I-frames initially, since they are easily (and quickly) parsed, and compressed domain techniques can be applied. This optimization can give several orders of magnitude improvement in performance in some circumstances, but since other libraries hide the structure of the MPEG bit stream from the programmer, this optimization cannot be used. In Dali, this optimization is trivial to exploit. The programmer can use the `MpegPicHdrFind` function to find a picture header, `MpegPicHdrParse` to decode it, and, if the type field in the decoded header indicates the picture that follows is an I-frame, call `MpegIPicParse` to decode the picture.

3.3 Implementation

Dali is currently implemented as a C run time library with approximately 50K lines of code. A Tcl binding is also available. It has been ported to Win95/NT, SunOS 4, Solaris, and Linux. The Dali library is divided into several packages according to their functionality and data type support. Supported data type includes PNM, GIF, JPEG, WAV, MPEG-1 and AVI. The code can be downloaded from `http://www.cs.cornell.edu/dali/`

One might wonder whether the layered architecture of Dali has any negative impact on performance. To answer this question, we compared three programs written in Dali to similar programs widely used in the research community. These benchmarks include the Berkeley MPEG decoder, the IJG JPEG encoder, and a use of the NETPBM toolkit. Our results show that Dali performs as well as these programs or better.

3.3.1 NETPBM

To compare Dali with NETPBM, we used the following task: convert a 1600x1200 GIF image to a 320x240 gray scale image. On a Sparc 20 workstation, the command `giftopnm input.gif | ppmtopgm | pnmscale 0.2 > /dev/null` takes 2.6 seconds. The Dali program that performs the same function takes 1.5 seconds and is about 110 lines long.

The Dali program performs better because the implementation of NETPBM is not optimized and overhead is incurred when data is piped from one program to another. These shortcomings could be addressed by writing a single C program that combines code from giftopnm.c, ppmtopgm.c, and pnmscale.c, but this is a time-consuming task. In contrast, the Dali program to perform the task can be easily optimized. For example, since Dali exposes the color table of a GIF image (as an `ImageMap`), we can perform the RGB-to-gray conversion on the color table instead of the RGB image. This modification improved the performance by 13% and required changing four lines of code.

3.3.2 MPEG decoder

We compared Dali with the Berkeley MPEG decoder (`mpeg_play`). Our full function MPEG to PPM converter required about 150 lines of Dali code. On a Sparc 20 workstation, the Dali program ran about 10% faster than the `mpeg_play` on a large variety of streams. We believe that Dali's specialized primitives for decoding I, P, and B frames contributes to the performance gain.

3.3.3 JPEG encoding

Dali JPEG encoding performance is comparable to the Independent JPEG Group's encoder (cjpeg). The IJG encoder will compress a 1600x1200 PPM image in 1.0 seconds of CPU time on a Pentium II 266 MHz WinNT workstation with 64MB of memory. The straightforward version of the equivalent Dali encoder, which reads the whole PPM image into three `ByteImage`s, converts them into the YUV color space, performs the DCT, and encodes the result, takes about 20% longer. We believe that the data copies associated with de-multiplexing the RGB data in the I/O buffers into the `ByteImage`s is responsible for the lesser performance of our version.

Not satisfied with this result, we rewrote the Dali encoder to divide the `ByteImage`s into horizontal strips using Dali's clipping mechanism. We then performed color-space conversion, DCT, and bitstream encoding on each strip separately. This design gives superior caching performance. The improved version of JPEG encoder also takes 1.0 seconds to encode the image.

These experiments show that the design principles that we adopted for Dali do not hurt performance. Rather, they allow flexible, optimized programs to be constructed with minimal effort.

3.4 Conclusion

The multimedia research community has traditionally built software from scratch in C or by using high-level libraries. We believe that neither approach is satisfactory. We therefore developed Dali, a software library for high-performance multimedia processing that provides lower level abstractions than most libraries, but much higher level than C. Dali is designed for high-performance, and is based on several design principles such as explicit resource management, resource sharing and thin operations.

We described Dali through examples and presented the design principles that make Dali a high performance library. These design principles can be used as a case study in designing high performance media processing library. We think that Dali is a useful tool to the research community because it allows efficient, processing-intensive multimedia software to be built with relatively small effort, and provides a vehicle through which research groups building this software can exchange their results.

Dali is being used as the processing engine inside our Degas media gateway. We discuss the design and architecture of Degas, along with the programming model in the next chapter.

Chapter 4

Extending Media Gateways with Customized Transformations

In this chapter, we present the design 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 processing on RTP-based video and audio streams.

Figure 4.1 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 [50] and ivs [77] 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 (called `degasclient`) is a modified version of vic. It is extended with the abilities to communicate with the gateways and has a user interface to select and use a deglet. The client first requests a service from

[1]Named after French impressionist Edgar Degas.

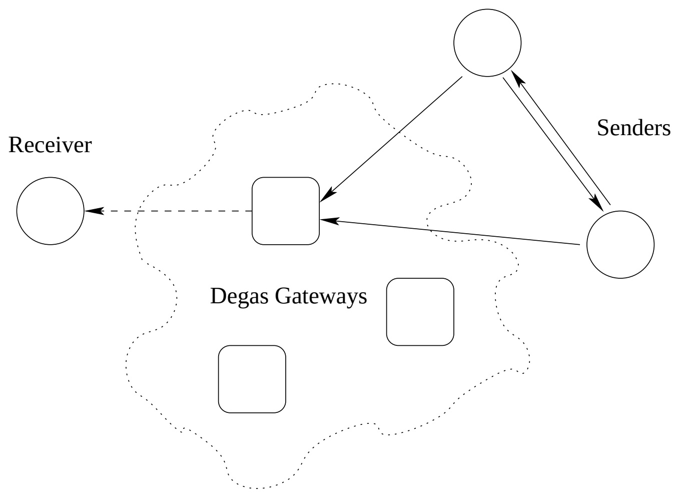


Figure 4.1: Degas system.

Degas. Degas selects a gateway to service the request. The client then uploads its deglet into this gateway. The gateway joins the session requested by the client. Input video streams are processed according to the specification in the deglet, and the resulting 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 the user to interact with the deglet and the gateway, for example to reconfigure the deglet, send user interface events (say, mouse clicks), or migrate the deglet to another gateway. The gateway can use the same control channel to send error messages back to the client for debugging. Figure 4.2 shows an example output stream produced by a Degas gateway.

This chapter describes how we extend a media gateway with user customized operations. We first explore the possible design space in Section 4.1. We then explain how a deglet can be written in Section 4.2. The optimization and execution of deglets are described in Section 4.3. We provide performance data in Section 4.4. Finally, we conclude in Section 4.5.

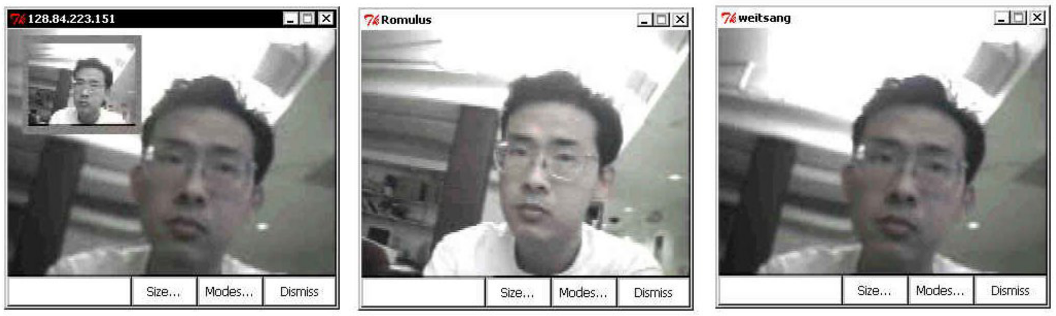


Figure 4.2: Output (left) and input streams (middle/right) of a Degas system.

4.1 Design Goals

We designed Degas to be flexible and easy to use. These two goals differentiate our system from other related research projects. We compare the existing systems with our work using these two metrics: *flexibility*, which measures the degree of freedom that end users have in customizing the system, and *simplicity*, which measures how easily the customization can be performed.

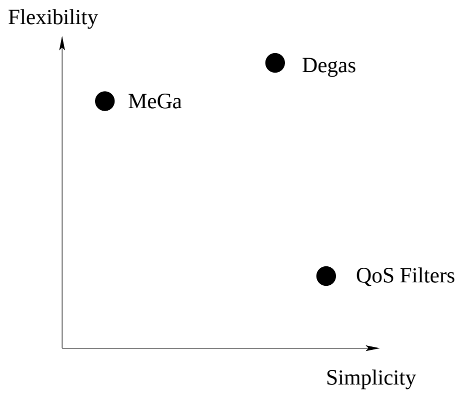


Figure 4.3: Design space of programmable media gateways.

Figure 4.3 shows the design space of current programmable media gateway systems. As described in Section 2.4.2, the work on QoS Filters [85] by Yeadon et al. provides a fixed set of media transformations for the users to choose from.

Although users can configure these filters via a graphical user interface (simple), they cannot extend the system with their own filters (inflexible). MeGa and Active Services [5] allow users to upload new transformations in the form of a service agent (flexible). However, users have to write the service agent in OTcl using Mash objects (complex). Furthermore, users are limited to transformations implemented inside Mash. Adding a new type of transformation, for instance, to change brightness, would require implementing the transformation in C++, recompiling Mash, and reinstalling the gateway. The installation of gateways is usually restricted to administrators and not available to the end users.

Our goals are to build a media gateway that has a high degree of extensibility and that is simple to extend. These two desired properties are highly dependent on the programming model of deglets. A deglet must be easily specified, yet powerful enough to perform useful operations on media streams. We discuss our programming model for a deglet in the next section.

4.2 Programming Model

The main consideration in selecting a programming model for deglets is simplicity while retaining flexibility and power. We want to make deglets 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 [55] for compatibility with the toolkits we used to build Degas, namely, Dali and Mash. We also chose to use a declarative model for programming deglets. A declarative model lets the user specifies what to do, but not how to do it. The user should not be concerned 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 formats 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 the properties of their streams in the middle of a session. The underlying execution engine determines the best way to perform a task.

We expose a single abstraction that users can manipulate: a frame from a video stream. Users can specify a sequence of operations on a frame. Our API hides the encoder and decoder operations, as well as the underlying representations from the users. This allows a deglet to be written in relatively few lines of code, yet allows flexible customization to be performed on the operations inside the gateway. Using this model, we believe we have achieved our goals of simplicity and flexibility.

We made some tradeoffs in the design. To achieve the goal of simplicity, we restricted the computations that can be performed on the video streams. We chose to provide extensibility by allowing users to compose a program using a fixed set of APIs. Users cannot define their own frame processing operations, nor can they extend the gateway with new video formats. However, our design allows Degas to be updated with new operations easily. The frame operations are currently implemented as dynamically loadable Tcl packages. We can include mechanisms to download new packages that implement new or updated versions of frame operations from a trusted authority. This would greatly simplify the deployment and maintenance of the gateways.

4.2.1 Examples

To better understand how a deglet is written, we present two examples in Figure 4.4 and Figure 4.5. We explain these two examples in detail in the rest of this section.

Figure 4.4 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 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 $\times$ 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.

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 streams from other sources are suppressed. For simplicity, we

```
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 4.4: A simple deglet.

assume that the number of sources is always larger than two. We explain this deglet below.

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 of 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 29 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 4.1, 4.2, 4.3 respectively. These lists are by no means complete, as we plan

```
sources {*}
num_of_sources {*}
input_video_session {224.4.4.4/4444}
input_audio_session {224.4.4.5/4444}
output_format {H261}

init_callback { outf } {
  set w2 [expr [frame_get_width $outf]/2]
  set lf [frame_clip $outf 0 0   $w2 [frame_get_height $outf]]
  set rf [frame_clip $outf 0 $w2 $w2 [frame_get_height $outf]]
  set prev 0
  set curr -1
}

talk_start_callback {src_id} {
  if {$src_id != $curr} {
    set prev $curr
    set curr $src_id
  }
}

recv_frame_callback { src_id inf outf } {
  if {$src_id == $curr} {
    frame_copy $inf $rf
  } else if {$src_id == $prev} {
    frame_copy $inf $lf
  }
}

destroy_callback {} {
  frame_free lf
  frame_free rf
}
```

Figure 4.5: A more elaborate deglet example.

Table 4.1: A summary of keys in deglet specification.

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.
preconditions	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 4.2: A summary of available callbacks 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 4.3: A summary of available frame operations 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`).

to add more operations to Degas. In particular, it would be interesting to add vision-related routines such as face detection and object tracking, thus allowing the possibility of encoding different regions in a frame with different qualities.

4.2.2 Preconditions for Choosing Gateways

Besides the key-values pairs described above, a deglet may contain a set of *preconditions*. The purpose of preconditions is to allow users 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. Possible 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 4.6. This test restricts gateways to those in domain *.cornell.edu, or gateways that are within 500 ms away. Many other tests are possible. 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, gateways below a certain price can be selected.

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

Figure 4.6: An example of using preconditions.

When a client requests a 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. This process will be discussed in greater detail in the next chapter.

As illustrated in the examples above, a deglet is a high-level, declarative style specification. They are short and simple to write. One of our more complicated deglets creates a "task bar" of incoming video streams, and lets users maximizes or minimizes a video stream by clicking on the task bar. This code is less than 100 lines. 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 engine is responsible for translating these specifications into optimized low-level code. We describe the execution of deglets next.

4.3 Execution of Deglets

The Degas execution engine is responsible for parsing the deglet specification and for efficient execution of the callbacks. The execution engine must recognize optimizations and translate the high-level API into appropriate low-level code. For instance, in Figure 4.4, 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 engine should simply copy the streams without decoding it.

We use Dali as our target for the translator. The execution engine is just a Tcl interpreter extended with Dali commands. Since Dali is designed with high-performance in mind, it gives programmers (or in our case, the translator) the flexibility of writing highly optimized programs. Also, Dali gives the programmers full control over memory usage and I/O. These features make Dali ideal for forming the basis of our execution engine.

The optimizations and executions are carried out as follows. We define 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. Figure 4.7 shows the execution engine of Degas.

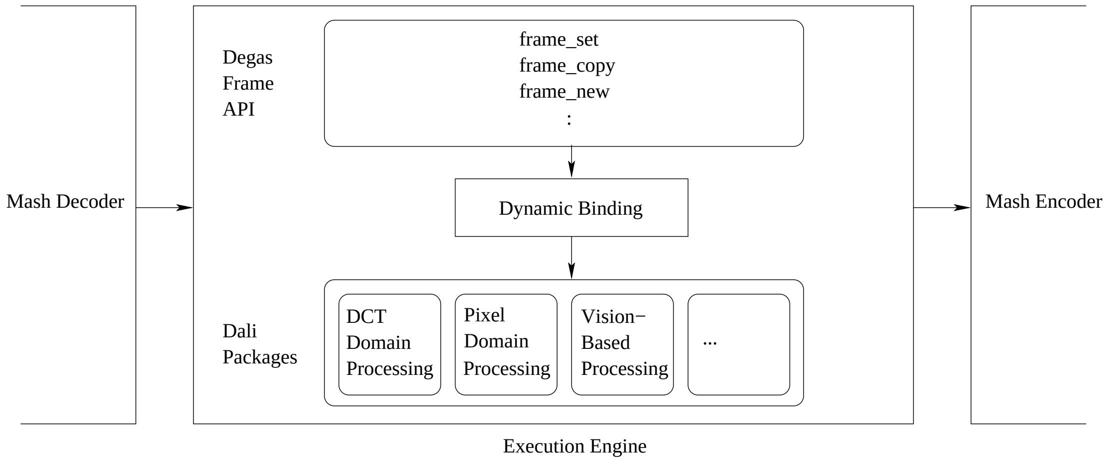


Figure 4.7: Execution engine in Degas.

Figure 4.8 shows some examples of the optimized operations and how binding is done. Line 1 to 11 are the definition of the `frame_copy` operation. It checks if a low-level method for copying frames with the same properties as the input source and destination already exists. If not, it generates the required method (line 8) and evaluates it (line 10). Line 12 to 24 show part of the method for binding the high-level API to low-level Dali procedures. In particular, line 18 to 21 compare the dimension of the source and destination. If the source is twice the size of the destination, we bind frame_copy to the low-level Dali procedures that shrink an image by half for the particular type and color decimation. Line 25 to 32 are examples of low-level Dali procedures for shrinking an image by half. The first procedure (line 25 to 29) shrinks color images while the second one (line 30 to 32) operates on gray scale images.

```
proc frame_copy {src dest} {
  set sw [frame_get_width $src]
  set dw [frame_get_width $dest]
  set sh [frame_get_height $src]
  set dh [frame_get_height $dest]
  set type [frame_get_type $dest]

  if {[info commands frame_copy($type,$sw,$dw,$sh,$dh] == ""} {
      bind_frame_copy_proc $src $dest $type $sw $dw $sh $dh
  }
  eval frame_copy($type,$sw,$dw,$sh,$dh) $src $dest
}

proc bind_frame_copy_proc {src dest type sw dw sh dh} {
       :
  set ssh [frame_get_hsubsample $src]
  set ssv [frame_get_vsubsample $src]
  set dsh [frame_get_hsubsample $dest]
  set dsv [frame_get_vsubsample $dest]

  if {($sw >> 1) == $dw && ($sh >> 1) & == $dh} {
     proc frame_copy($type,$sw,$dw,$sh,$dh) { src dest } \
        [info body \
            shrink_2x2($type,[expr $ssh/$dsh],[expr $ssv/dsv])]
  } elseif {
      :
}

proc shrink_2x2(yuv,1,2) {src dest} {
   byte_shrink_2x2 [frame_get_y $src] [frame_get_y $dest]
   byte_shrink_2x1 [frame_get_u $src] [frame_get_u $dest]
   byte_shrink_2x1 [frame_get_v $src] [frame_get_v $dest]
}

proc shrink_2x2(gray,1,2) {src dest} {
   byte_shrink_2x2 [frame_get_y $src] [frame_get_y $dest]
}
```

Figure 4.8: Optimized Dali code and dynamic bindings of frame_copy.

4.4 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 [52] 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. This experiment confirmed our belief that the overhead in optimizing is small. 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 increased significantly to 43.8 ms.

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 4.4. The table shows that the delay introduced by the decode-process-encode pipeline is reasonably small.

Table 4.4: Latencies introduced by the decode-process-encode pipeline and the CPU load incurred for different deglets.

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

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 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 4.4). 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 4.9. 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 4.10(a) and Figure 4.10(b) show that Degas gateway introduces some jitter, but are within a tolerable range (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.

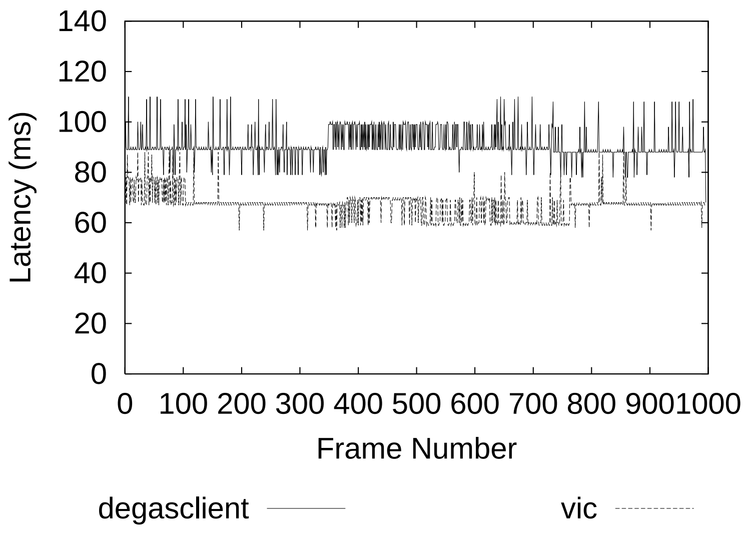


Figure 4.9: End-to-end delay between the sender and the receiver.

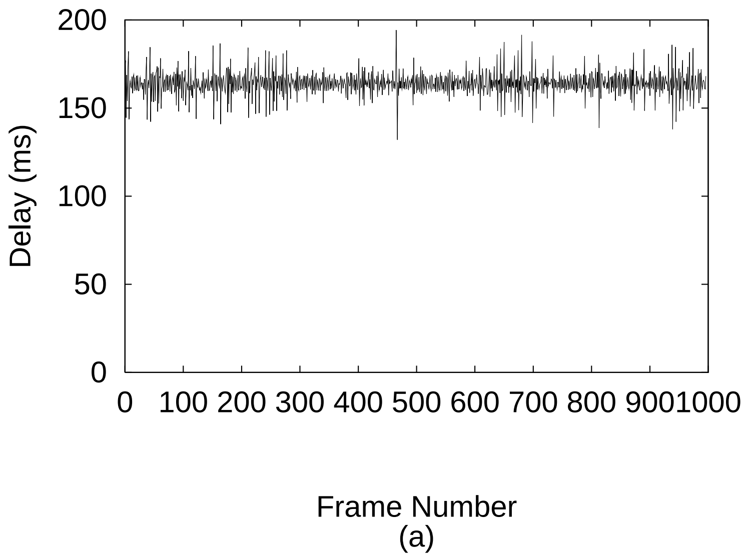
The reason is that we have not implemented any resource management in Degas yet.

4.5 Conclusion

This chapter describes the programming model of a flexible and extensible media gateway system, called Degas, which allows users to request customized processing on media streams. We designed Degas to be efficient and simple to use, while being compatible with existing popular MBone tools.

Degas is implemented using C++ and the Mash toolkit. A preliminary prototype is available from our web site at `http://www.cs.cornell.edu/degas`.

This prototype programmable media gateway serves as a platform where many reseach issues can be explored. One of the issues that we have been studying is how to locate an appropiate gateway for serving a client's request. We have developed a decentralized, adaptive protocol to solve this problem. We describe this protocol in the next chapter.



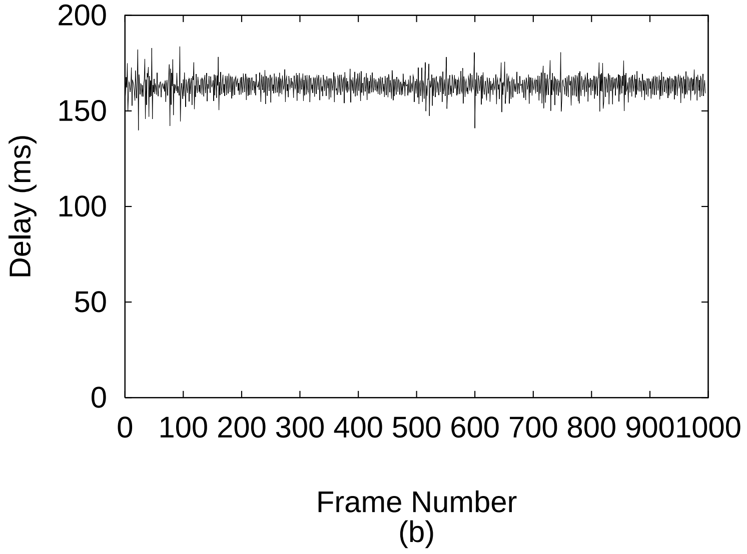


Figure 4.10: (a) Inter-frame rendering delay using Degas. (b) Inter-frame rendering delay without Degas.

Chapter 5

An Adaptive Protocol for Locating Degas Gateways

In this chapter, we present a solution to choosing an appropriate gateway for running a deglet requested by some user. Running the deglet on a gateway that is strategically located in the network could use network bandwidth more efficiently. For example, if the output video stream requires lower bandwidth than the input stream, then the deglet should be run on a gateway that is close to the sender. On the other hand, if the deglet outputs a higher bandwidth stream, the deglet should be run close to the receiver.

The problem of determining the best gateway is an optimization problem. The dynamic nature of the network prevents us from solving the problem using a centralized, combinatoric algorithm. Senders and receivers may join and leave video sessions, new gateways may be added and deleted, and the underlying network behavior changes continuously. Therefore, we opted for a distributed, adaptive algorithm in Degas.

We designed an application-level protocol, called the Adaptive Gateway Location Protocol (AGLP), for choosing a gateway that minimizes bandwidth consumption. Although we designed AGLP to work with Degas, we believe that it can be modified to suit other applications as well.

AGLP is a soft-state protocol based on the announce-listen model widely used in MBone tools. The simplicity of the model allows us to build a scalable, robust protocol that is resilient to crashes and message loss. AGLP adapts to changing network conditions, as well as the birth and death of gateways, senders, and receivers, by migrating deglets (also called *services*) between gateways. An additional requirement on our protocol is that it assigns a new service to a gateway rapidly.

We designed our protocol to be compatible with existing MBone tools. No changes are required at the senders. This means that traditional MBone tools such as vic [50] and ivs [77] can be used as the video sending application. This makes it possible to deploy our protocol without affecting the existing MBone community.

Our simulations support that AGLP achieves its goals of rapid assignment and adaptive placement, while keeping the load on the network low. The rate of migrations is small, and a good gateway for such a migration can be selected within a minute.

The rest of this chapter is organized as follows. We describe the AGLP protocol in Section 5.1. We analyze the performance in Section 5.2. We briefly address some implementation issues in Section 5.3 and conclude this chapter in Section 5.4.

5.1 Protocol Description

Before we describe our protocol, we present the symbols and terminology used in our description:

- g is a well-known multicast channel used for exchanging control messages among the gateways, and between the gateways and client. Every gateway and client listens to g.
- s is a multicast session.
- C is the client that requests some processing to be done on video streams.
- $G_0, G_1, .., G_m$ are gateways available for running a deglet requested by C. One of the gateways will be selected to service C. Without loss of generality, we let G_0 be the current gateway servicing C.
- $S_0, S_1, .., S_k$ are video senders participating in video session s. These senders can be normal MBone video sources. They need not be aware of the existence of the gateways or C.

AGLP uses propagation time as a parameter to decide whether a gateway is suitable to service a client. It does not take geographical locations, topology or number of hops into consideration. Previous study [7] shows that there is little correlation between these parameters. We use propagation time as this directly corresponds to end-to-end delay, a parameter we care about.

For simplicity, we assume that each client can submit only one deglet at a time, and each deglet can read from only one session. We also assume that all participants run the network time protocol NTP [52], which we rely on to measure the propagation delay of a packet. However, in the absence of NTP, we can use other schemes to estimate propagation delay, such as SPAND [68], or simply ping.

Our protocol consists of two phases (see Figure 5.1). The first phase, *Quick-Start Phase*, chooses a gateway G_0 that is close to C, without worrying about optimizing bandwidth consumption. The second phase, the *Adapting Phase*, minimizes the bandwidth consumption by migrating services to a better gateway. We describe these two phases in Section 5.1.1 and Section 5.1.2 respectively.

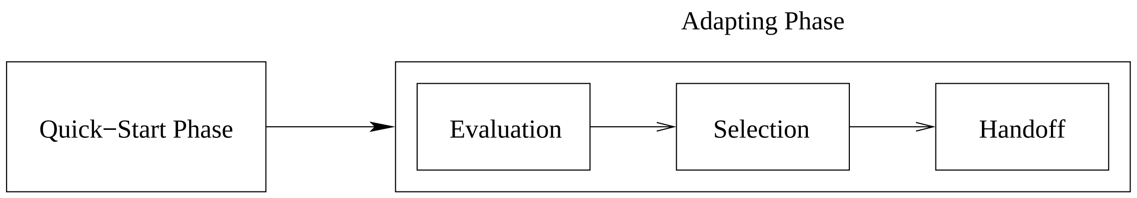


Figure 5.1: Different phases in AGLP.

5.1.1 Quick-Start Phase

There are two reasons why the Quick-Start Phase is necessary. First, we want to reduce the start-up latency experienced by the user. Secondly, we do not have any knowledge about the behavior of the deglet requested by the client, nor do we know anything about the session (such as the identity of the senders, and bandwidth of incoming video streams). The gateway we select at the Quick-Start Phase serves as a temporary gateway. This gateway collects information so that further optimization can be done. The Quick-Start Phase works as follows (see Figure 5.2).

The client C who wants to request some processing to be done on the gateway first multicast a request message onto the common multicast channel g. The request message contains the preamble section of a deglet. Specifically, it contains the

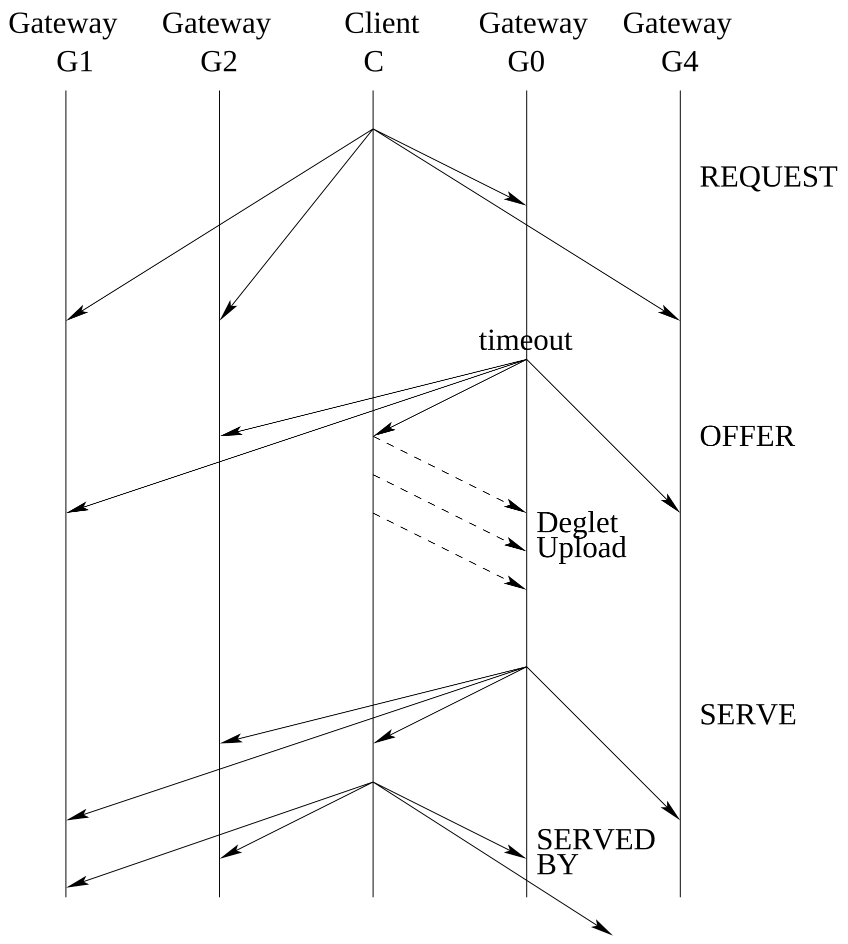


Figure 5.2: The Quick-Start Phase of AGLP.

address of the input session s and a set of preconditions. Each gateway evaluates the preconditions to see if it is eligible to service the client C.

A gateway G_i that is available and eligible to serve C replies with an offer(C) message. However, instead of replying immediately, we employ multicast damping (Section 2.3.2) to reduce the number of offer messages received by C. Each G_i waits for time $T_{\text{offer,i}}$ before multicasting the offer onto g. Moreover, a gateway will suppress its offer(C) message if it has received an offer(C) message from another gateway while waiting.

Client C listens to g and accepts the first offer that it receives. Without loss of generality, let the first offer that C receives be from gateway G_0. C subsequently creates a TCP connection with G_0 at port p, where p is a port number embedded in the offer(C) message. Subsequent offers from other gateways will be ignored by C. C sends the deglet, along with the multicast address of the output session s', to G_0 using the TCP connection.

After G_0 has received all the necessary information, G_0 joins the session s, processes incoming video streams, and multicasts the output onto channel s'. C listens to channel s' to receive the post-processed video it requested. At this point, we enter a state where gateway G_0 is serving client C. G_0 and C periodically announce this relationship onto g. Every T_{serve} seconds, G_0 announces a serve(C) message onto g. Similarly, C sends a served-by(G_0) message to G_0 every $T_{\text{served}-\text{by}}$ seconds.

The receipt of serve(C) message by C indicates that the Quick-Start Phase has completed successfully. If C does not receive any serve(C) message in a period of length T_{request}, C will restart the whole process by sending another request message. Otherwise, the Quick-Start is successful, and AGLP proceeds to the next phase.

5.1.2 Adapting Phase

During the Adapting Phase, a service for C may be migrated from the current gateway G_0, to another more suitable gateway, as more information about the session is discovered and changes in the environment are detected. The Adapting Phase consists of three stages: evaluation, selection and replacement (see Figure 5.3 for an example). In the evaluation stage, each gateway evaluates itself against G_0 to check if it is more suitable than G_0 for serving C. Once a gateway determine that it can serve G_0 better, it will notify G_0. G_0 periodically runs a selection process, to select the best alternate gateway. Once a replacement G_r is chosen, G_0 hands-off the service for C to G_r. We explain these three stages in greater detail in the following subsections.

Stage 1: Evaluation

We first introduce a few variables that corresponds to the criteria used to perform evaluation:

- b_i: the bandwidth of video stream from sender S_i
- b_C: the bandwidth of the output video stream
- $d_{i,j}$: the distance between gateway G_i and sender S_j
- $d_{i,C}$: the distance between gateway G_i and client C

We now describe how this information is collected and how the evaluation is performed.

After joining session s, G_0 starts to collect information about the current session. This information includes the identity of the senders in the session, bandwidths of the input streams and the output stream, and the distance (or latency)

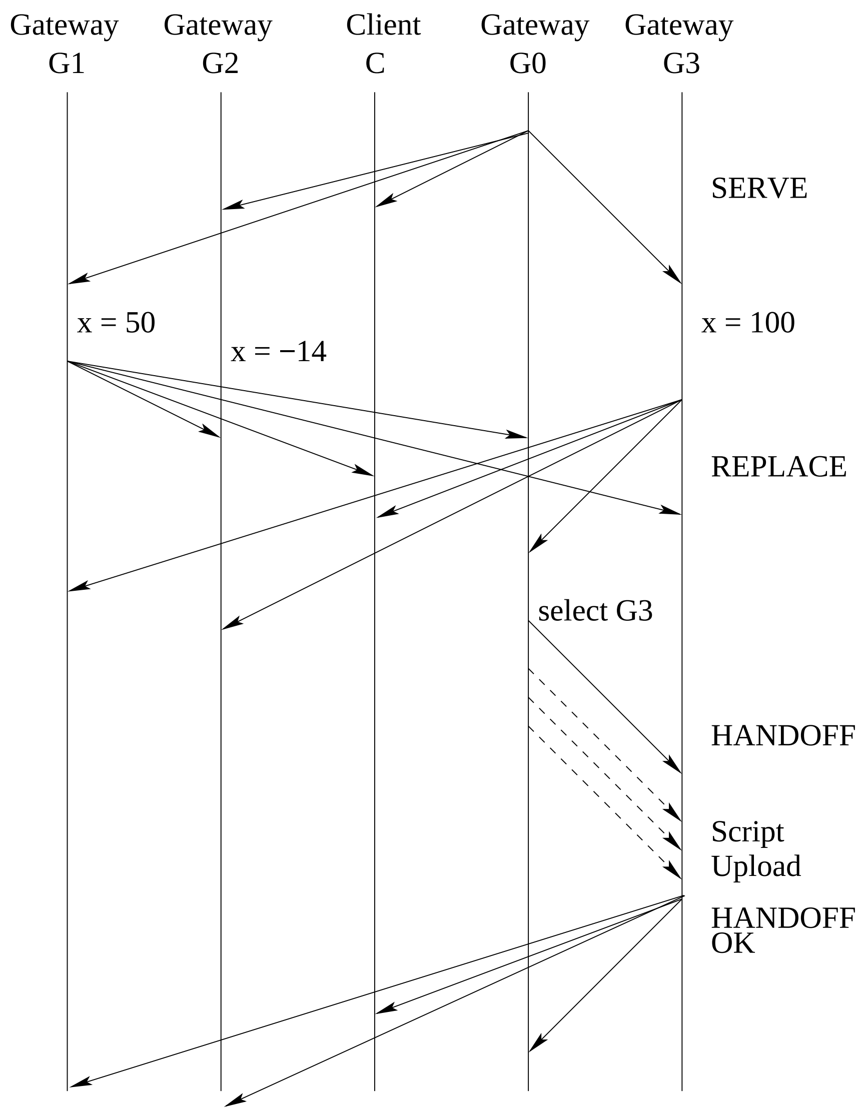


Figure 5.3: The Adapting Phase of AGLP.

$d_{0,j}$ from each sender S_j. This information can be learned from RTCP [67] packets. The identity of the senders can be found by looking at the source of a RTCP sender report. The sending bandwidth can be derived from the byte count included in a sender report. Finally, the distance can be calculated by comparing current time to the NTP timestamp embedded in a sender report. The information about the current session and the preamble section of the deglet are included in the serve messages and multicast onto group g.

Each gateway G_i, that is available and eligible to serve C, maintains a table of distances to itself from the sources, $D_{self} = d_{i,0}..d_{i,k}$. This table is maintained as soft-states, and is refreshed by periodically joining session s, and listening for RTCP packets. A distance can be calculated by subtracting the NTP timestamp of a sender report from the arrival time.

Each gateway, upon receiving a serve(C, s) message from G_0, starts the evaluation test to compare the suitability of serving client C. The test produces a *score*, x_i. This score is calculated as follows. First, let U_i be

$$U_i = \sum_{j=0}^{k}(b_j \times d_{i,j}) + b_C \times d_{i,C}$$

Intuitively, U_i corresponds to the bandwidth consumption. We calculate x_i as

$$x_i = U_0 - U_i$$

A score $x_i > 0$ indicates that G_i is better than G_0 for serving C.

Figure 5.4 shows an example on how the score is calculated. Suppose both S_1 and S_2 are sending streams at 32Kbps and our gateway is transforming these two input streams into a single stream at 48Kbps. The value of U_1 is $(32 \times 2 \times 0.03) + (48 \times 0.15) = 9.12$Kb and the value of U_0 is $(32 \times 2 \times 0.1) + (48 \times 0.05) = 8.8$Kb. Hence the score x_1 is -0.32.

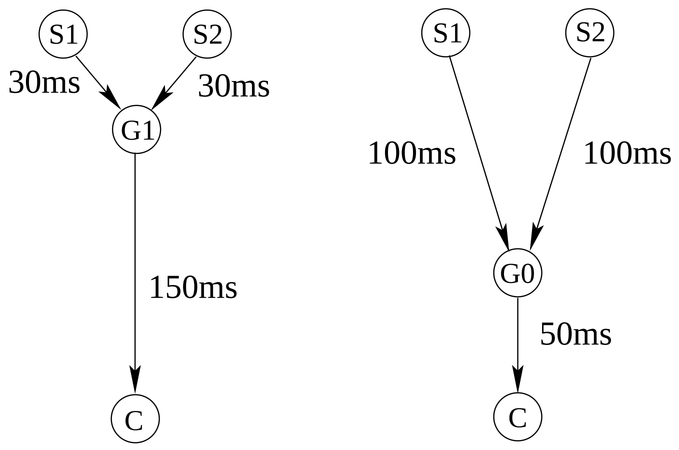


Figure 5.4: Example of score calculation.

Each gateway with a score larger than ϵ will try to replace the current gateway. We choose a threshold ϵ instead of 0 for two reasons. First, a score between 0 and ϵ indicates that the gateway is only slightly better than G_0. The small improvement that we gain is not worthy of the overhead caused by the replacement process. Second, by ignoring gateways that are only slightly better, we can avoid unnecessary oscillation caused by small changes in network conditions. We discuss how a replacement is selected by G_0 next.

Stage 2: Gateway Replacement

After evaluation, each gateway with a score larger than ϵ will notify the current gateway, and wait for a reply. This process is similar to the Quick-Start Phase. Again, we use multicast damping for scalability reasons. The gateways start a timer and wait for $T_{\text{replace},i}$ seconds. When the timer expires, gateway G_i multicasts a replace(C, x_i) message onto g. If G_i receives another replace(C,x_j) message from another gateway G_j and $x_j > x_i$, then G_i suppresses its own replace message.

The current gateway keep tracks of the gateway with the lowest score so far, which we call the replacement gateway G_r. T_{adapt} seconds after G_0 receives the first replace message, gateway G_0 unicasts a message handoff(C,p) to G_r. G_0 then establishes a TCP connection to G_r at port p through which G_0 sends the deglet to G_r. G_r subsequently starts the service, and multicasts a handoff-ok(C,G_0,s'') announcement, where s'' is a new multicast address where the processed media stream is going to be sent.

Stage 3: Service Handoff

G_r joins session s, starts processing the input video streams, and sends the output onto session s''. G_r also begins the periodical announcement of serve(C, s'') messages.

At this stage, both G_r and G_0 are providing service for C. Upon receiving both handoff-ok(C,G_0,s'') and serve(C,s''), C knows that another more suitable gateway has been found and this new gateway is ready to serve C. C can now switch from group s' to group s''. C stops announcing served-by(C,G_0) and starts announcing served-by(C,G_r). G_0 stops processing video streams from s eventually after no served-by(C,G_0) is received for T_{bye} seconds.

We provide a summary list of messages involved in this protocol in Table 5.1.

5.2 Analysis and Simulation

In this section we evaluate our protocol. In particular, we want to confirm that our protocol satisfies two desirable properties:

- robustness:

Table 5.1: A summary of message types in AGLP

request(d)	A request for service by a client. d is the preamble section of the deglet.
offer(C, p)	A response to a request message from client C. Indicates that the sending gateway is available to serve C. C should contact this gateway at port p for details.
serve(C, S, D, d)	The sending gateway is currently running a service for C. S is the list of session members, D is a vector containing distances from each member in S as well as the distance from C. d is the preamble section of a deglet.
served-by(G)	Response to the gateway serving C to notify that C is still listening to output from G.
replace(C, x)	Notify others that the sending gateway is more suitable for serving C. x indicates how much better the sending gateway is.
handoff(C, p)	Message from the current gateway to G' to indicate that G' has been chosen to replace the current gateway for serving C. G' should listen to port p for service specification.
handoff-ok(C, G, s')	Announcement from a new gateway G' that it is ready to replace G to serve C. s' is the new multicast address where the output from the service will be sent.

  - a gateway eventually runs the service requested by a client;
  - all services are eventually terminated when no client is listening;
  - the service is eventually moved to the optimal gateway.

- scalability:
  - as the number of gateways increases, the number of states maintained and the number of messages exchanged does not increase significantly.

5.2.1 Robustness

We achieve robustness by maintaining only soft-states which are periodically forgotten and need to be refreshed. Soft-state protocols are used in many light-weight protocols in MBone applications such as SDP [32] and RTCP [67]. Failure recovery is automatic in soft-state protocols, since the failure of a gateway or network link will cause refresh messages to be lost and states to be forgotten. Refresh messages in AGLP include serve and served-by—we illustrate how they support failure recovery by describing two scenarios below.

- Suppose that the gateway that is serving C crashes. The periodic serve message will cease and C will eventually forget that some gateway is servicing it. C will start requesting service again by entering the Quick-Start phase.
- Suppose that the message handoff-ok is lost on its way to C. C will not switch to the new gateway. Even though the new gateway has started serving C, it will not receive a served-by message from C. The new gateway will eventually timeout after T_{bye} seconds, and end its service.

We simulated AGLP in networks with up to 50% loss rate. Although this caused somewhat longer start-up/handoff latencies and redundant requests, the protocol still worked correctly.

5.2.2 Scalability — Memory Requirements

We envision that the number of gateways running in the network $|G|$ will be large (up to thousands), and the number of clients requesting service to be in the same range. The number of senders per client, $|S|$, however, is expected to be small (say, less than 10). Similarly, because the processing requested by client could be computation intensive, we expect the maximum number of clients that can be served by each gateway, $|C|$, to be small as well.

Each gateway maintains the following soft-states:

- A list of clients it is currently serving;
- The gateway with the best score so far for each client it serves;
- A table that records the distance to all senders for each client it serves (and available to serve, for evaluation purposes);
- A table that records the bandwidth of all input streams and output streams for each client it serves (and available to serve, for evaluation purposes).

On the client side, the only soft-states that are maintained are the sessions to listen to, and the gateway currently serving the client.

The size of the state maintained in the gateway is thus $O(|S| \times |C|)$, and is $O(1)$ for the client. Since a gateway does not keep state for every other gateway, and both $|S|$ and $|C|$ are expected to be small, our protocol is scalable in terms of memory size.

5.2.3 Scalability — Networking

Multicast damping is a widely used technique to improve scalability in one-to-many protocols (*e.g.*, it is used in IGMP [23] and SRM [24]). As described in Section 5.1, we use multicast damping for the request-offer and serve-replace message exchanges to avoid implosion of messages. The effectiveness of this technique, however, depends heavily on the timeout values chosen, T_{offer} and T_{replace}. Even though there is extensive work done in analyzing the effect of timers in multicast damping (see, for example, [24] and [53]), there are some unique requirements for our timers. T_{offer} should be proportional to the distance from the client, so that the first reply received by the client comes from the gateway that is closest to the client. For T_{replace}, the timer value should be inversely proportional to the score of a gateway. We discuss these two parameters in this section.

In order to evaluate the performance of AGLP under these parameters, we simulate our protocol using the ns2 network simulator and run it on a 500-node topology generated using the gt-itm toolkit [13]. We place gateways and the client at random locations in the generated network.

In AGLP, we set the value of T_{offer} to $k \times d$, where k is a constant and d is the propagation delay between gateway and client, measured using an NTP timestamp embedded in the request message. A small value of k results in a lower start-up latency, but a larger number of duplicates. The number of duplicates also depends on the distribution of gateways in the network. If gateways are sparsely distributed, then the number of duplicates increases.

We tried different values of k in our simulations. In Figure 5.5 we show the average number of duplicate offer messages received by the client for different values of k in cases where the number of gateways G is either 50, 100, or 200. A value of $k \geq 2$ causes the number of duplicates to stay below 3 even as the number of gateways increases up to 200. Figure 5.6 shows the latencies that the client experiences.

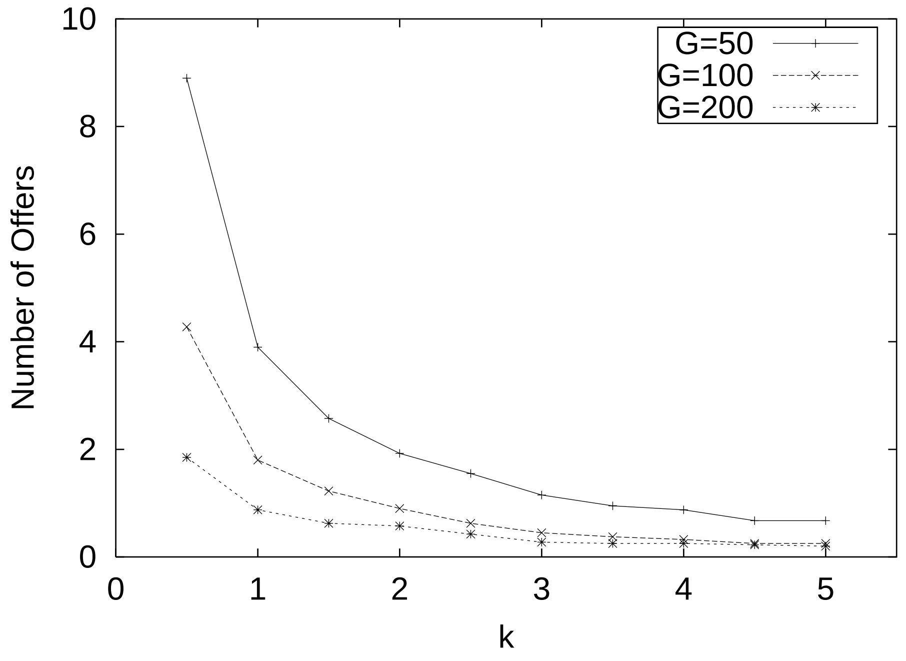


Figure 5.5: Duplicate offer messages for different values of k and G (the number of gateways).

We conclude that $k = 2$ works well in reducing the number of duplicates while keeping the start-up latency within a reasonable time. In Figures 5.7 and 5.8 we show the behavior of multicast damping as a function of the number of gateways in more detail, along with a 95% confidence interval for each measurement. Our experiments indicate that AGLP scales well for $k = 2$. In the remaining experiments we are using this value for k.

We set the value of T_{replace} to k'/x, where x is the score. In Figure 5.9 we show the number of duplicate replace responses as a function of k'. We see that for $k' > 200$ the number of duplicates is under 10, which we consider acceptable.

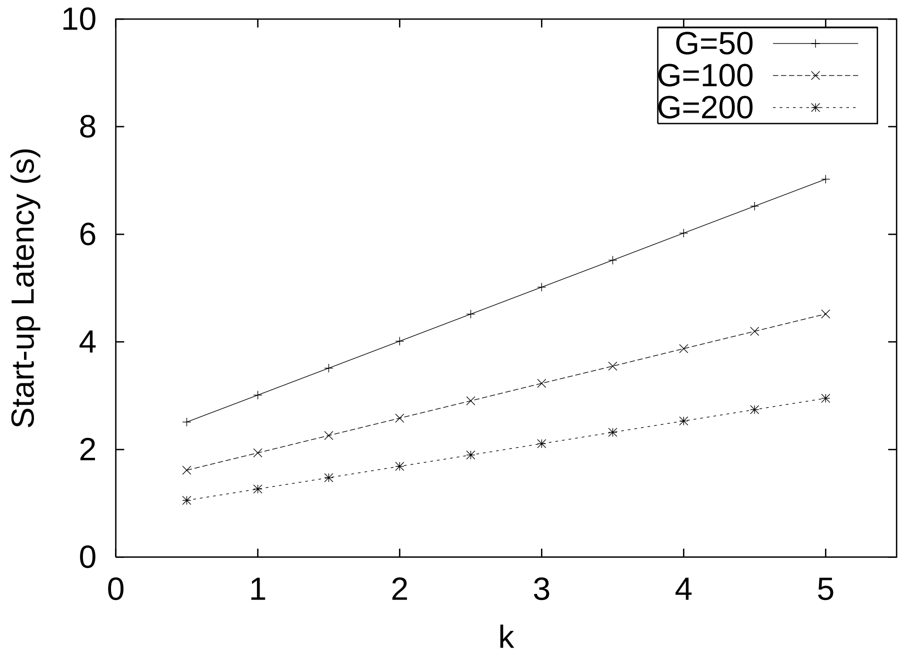


Figure 5.6: The delay between sending a request and receiving the first offer.

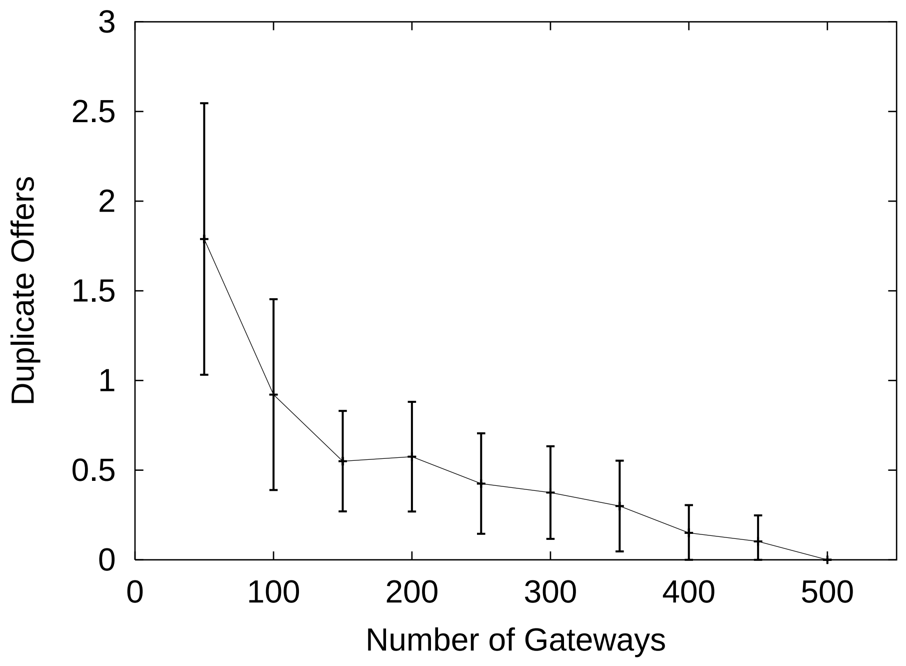


Figure 5.7: Duplicate offer messages for $k = 2$.

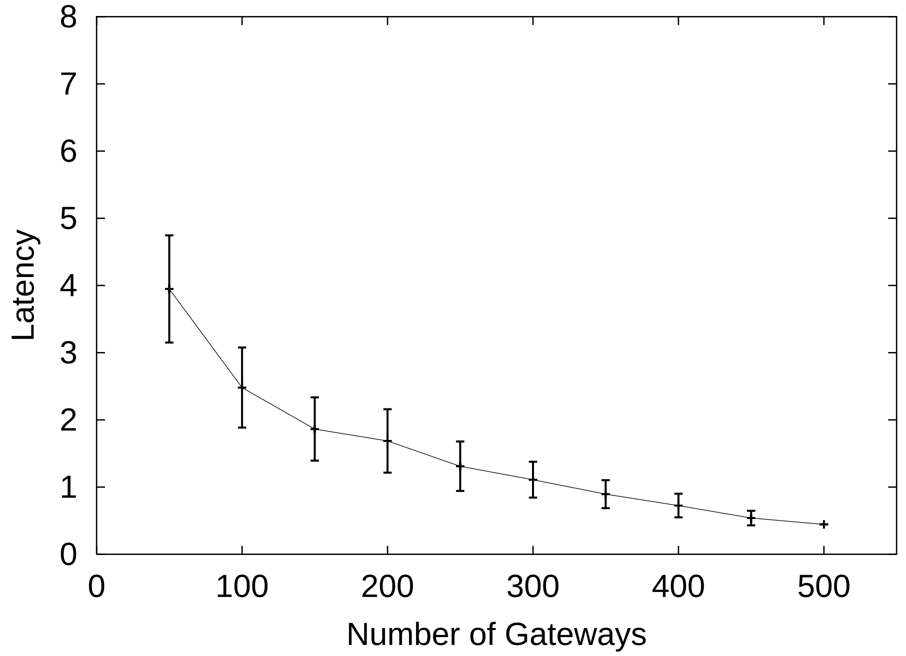


Figure 5.8: The delay between sending a request and receiving the first offer for $k = 2$.

Figure 5.10 shows that the average number of migrations before a service reaches the optimal gateway does not increase with k'. We were surprised by this result. After all, as k' goes up, it becomes less likely that the client will receive a response from the optimal gateway within T_{adapt}. However, after further consideration we are able to explain this.

After the current gateway gets the first replace response, it waits T_{adapt} seconds before selecting a gateway to hand-off to. That is, after sending the last serve message, it waits a total of $RTT_1 + k'/x_1 + T_{\text{adapt}}$ seconds, where RTT_1 is the round-trip time to the first responding gateway, and x_1 is the score at that gateway. In order for the optimal gateway's response to be received in time, we need to have the following condition (see Figure 5.11):

$$RTT_{optimal} + \frac{k'}{x_{optimal}} < RTT_1 + \frac{k'}{x_1} + T_{\text{adapt}}$$

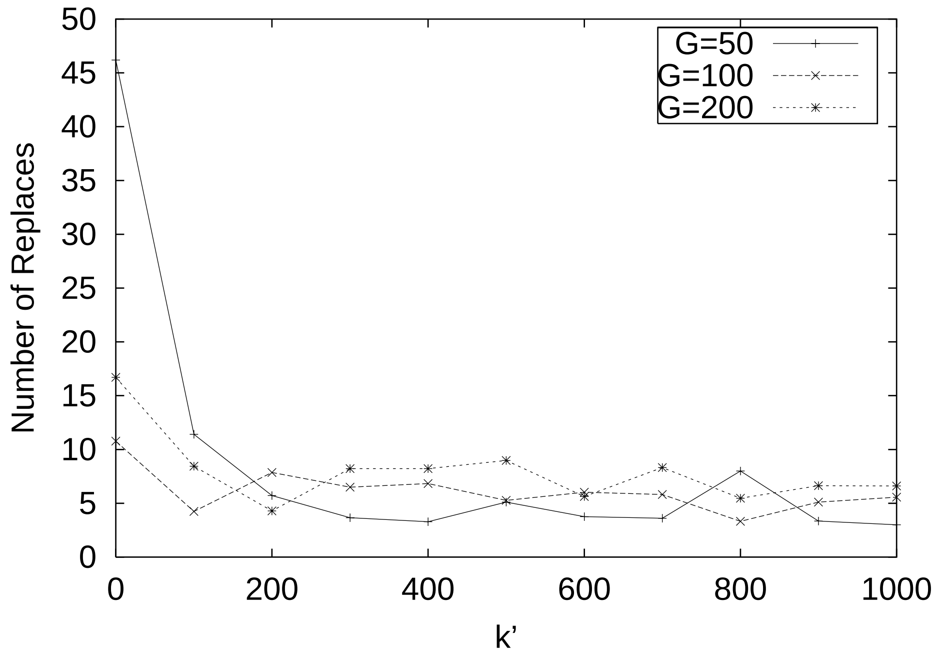


Figure 5.9: Duplicate replace messages.

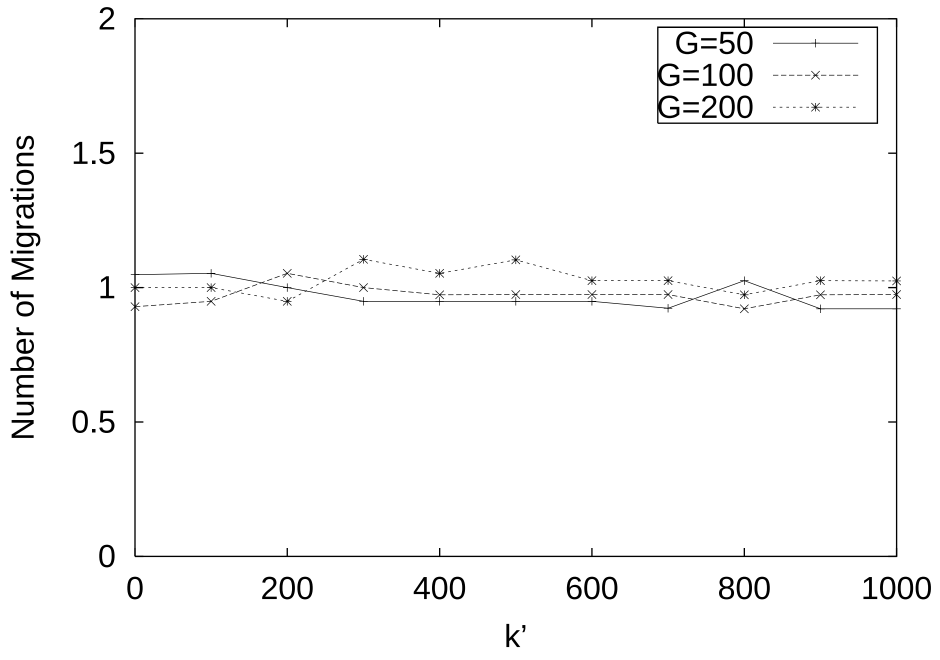


Figure 5.10: Number of migrations needed to migrate to an optimal gateway.

We can rewrite this as:

$$\frac{RTT_{optimal} - RTT_1 - T_{\text{adapt}}}{(\frac{1}{x_1} - \frac{1}{x_{optimal}})} < k'$$

Thus, increasing k' does not decrease the chances that the optimal gateway responds in time, as reflected in Figure 5.10.

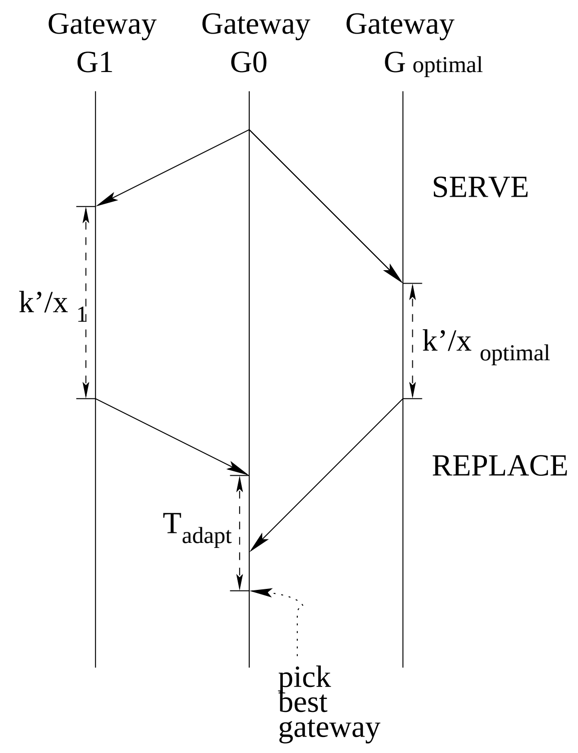


Figure 5.11: Exchanges of serve and replace messages.

In the following experiments, we use $k' = 1000$ as a conservative choice. For this value of k', we find that there are no more than 10 replace messages received (see Figure 5.12) even if we run a gateway on all 500 nodes in the network. There were at most two migrations in all runs of our simulations (see Figure 5.13 for averages and 95% confidence intervals). In Figure 5.14 we show how this translates into time. On average, all services were migrated to the optimal gateway within 40 seconds, which we find acceptable.

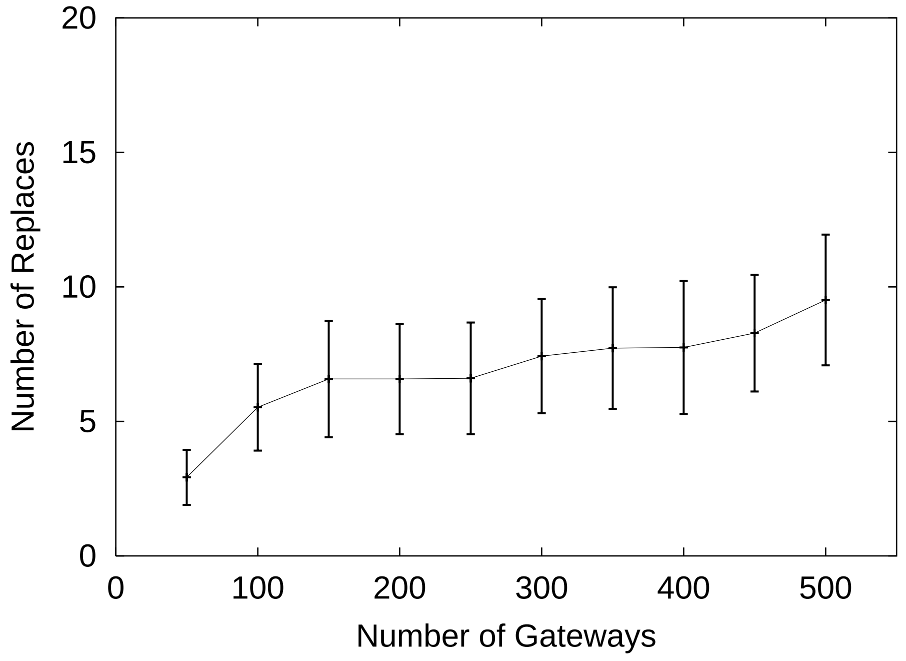


Figure 5.12: Duplicate replace messages received by a gateway for $k' = 1000$.

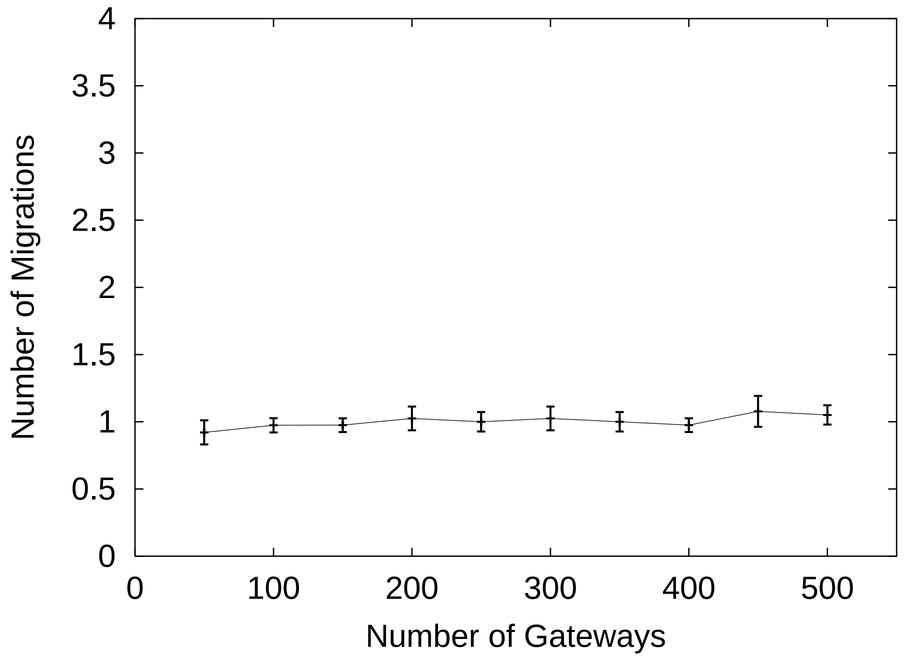


Figure 5.13: Number of migrations to migrate to an optimal gateway for $k' = 1000$.

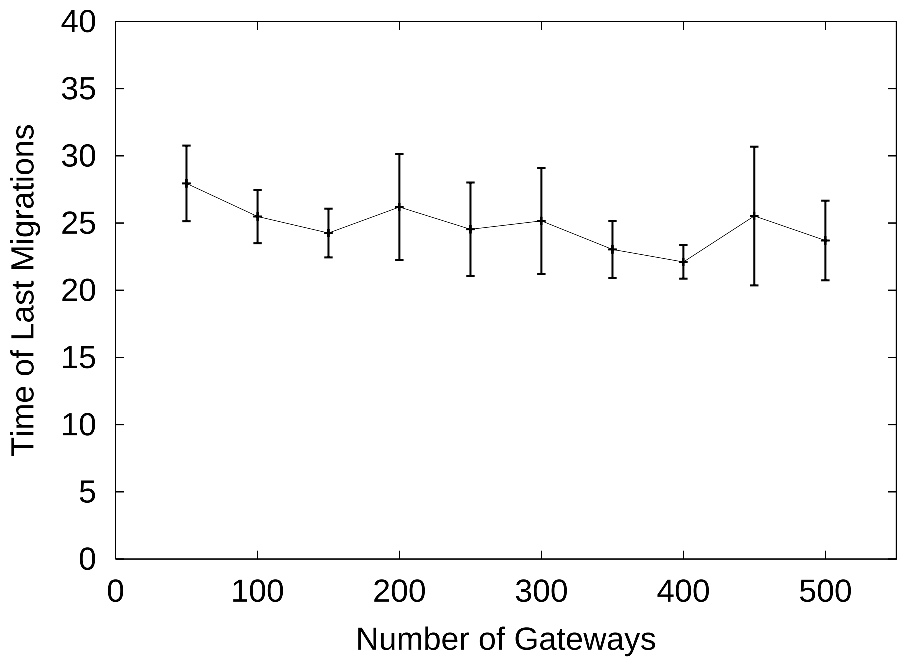


Figure 5.14: Time to migrate to an optimal gateway for $k' = 1000$.

5.3 Implementation

We have implemented AGLP in our Degas system using Mash. We made several improvements to the basic AGLP design to make AGLP more usable in practice.

- When a gateway starts running a deglet, there is an overhead in loading the Dali libraries. As a result, we found that the measurements of propagation time are not accurate for a short time after a deglet starts running. To counter this behavior, we do not send out session information along with serve messages for a brief period of time (30 seconds in our implementation).

- The propagation time between two hosts can vary greatly in practice, depending on load in the network. This causes our scores to fluctuate and results in unnecessary migrations. We modified AGLP to take a weighted average of the previous scores to smooth out the scores. Additionally, we migrate a deglet only when scores are larger than threshold ϵ consecutively for a certain number of times (4 times in our implementation).

5.4 Conclusion

In this chapter, we presented an adaptive control protocol called AGLP for running services on media processing gateways in the Internet. Our protocol supports the following functionality:

- allowing the client to request a service, and submit a deglet to a gateway;
- deciding which gateway should be used to perform a service;
- migrating services to more suitable gateways (adaptability).

AGLP builds on the announce-listen paradigm and uses soft-states to maintain information. As a result, our protocol is both scalable and robust. AGLP is compatible with existing MBone tools, so that no changes are required at the senders. Furthermore, the existence of gateways and clients is transparent to the senders.

Although AGLP is designed for the Degas system, the protocol can be modified for any application that needs to decide where to run certain services inside the network. With the increasing interest in the research community to move computation, traditionally performed at the edge of the network, into the network itself, we believe applications for AGLP will increase in the future.

Chapter 6

Distributing Deglets Over Multiple Degas Gateways

In Chapter 5, we discussed how a deglet can be assigned to a single gateway for execution. In this chapter, we describe a mechanism to partition a deglet into multiple sub-deglets for execution on multiple gateways.

Amir et al. introduced the notion of *composable services* for media gateways in [5]. By flowing through multiple gateways, multiple operations can be performed on a media stream before it reaches the receivers. This in effect creates a data-flow pipeline on the streams.

To clarify this approach, consider an operation that transcodes a H.261 video into MJPEG format and scales the frame size by half. This can be divided into two operations, one that transcodes the video, and another that resizes the video streams (see Figure 6.1(a)). These two operations form a linear pipeline. More complicated pipelines, in the form of a tree, are also possible. Consider an operation that creates a "quad-splitter" view by scaling four video streams, and merging them into one output stream. Such an operation would be useful, for example, for previewing what is being shown on multiple multicast channels. This operation can be performed on five gateways – four to scale the video streams, and one to combine the outputs from the first four gateways into a "quad-splitter" view (see Figure 6.1(b)).

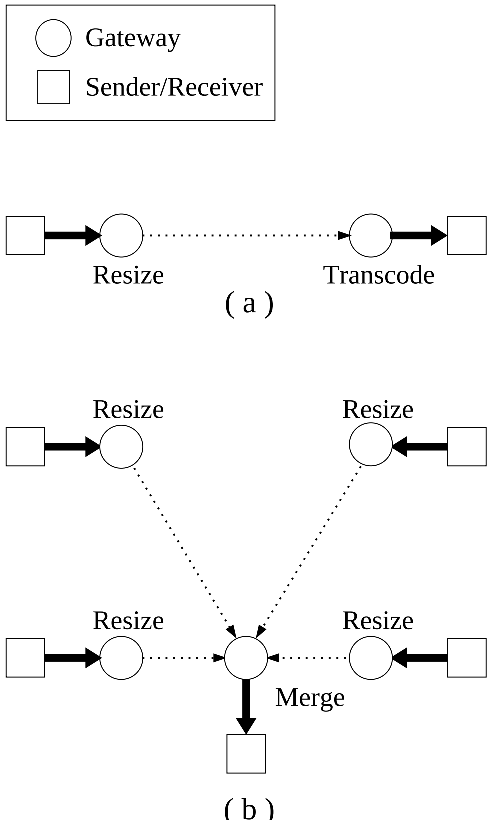


Figure 6.1: Examples of composable services.

There are several advantages in using multiple gateways to service a media stream, as opposed to using a single gateway. First, computation load can be better distributed among the gateways. This can lead to higher throughput and better load balancing. Second, by transforming media streams at appropriate locations, we can reduce bandwidth consumption. For instance, in the examples described above, the scaling operations are performed near the sources, and merging and transcoding are performed near the receivers, thus minimizing the amount of data that is sent into the "middle" of the network. Finally, it is possible for output from a sub-deglet to be shared by different users requesting different services. For example, if the transcoding service in Figure 6.1(a) shared a common source with the "quad-splitter" service, then the output from the scaling gateway could be shared by both services.

In this chapter, we present our preliminary work on composable services in media gateways. The rest of the chapter is organized as follows. Section 6.1 details our service model. Section 6.2 describes some constraints of our model. Section 6.3 presents the core problem that we wish to solve. Section 6.4 describes an algorithm for decomposing deglet into sub-deglets. Section 6.5 describes how we extend AGLP to locate gateways and assign sub-deglets to them. We present performance results of our protocol in Sections 6.6 and 6.7, and conclude in Section 6.8.

6.1 Service Model

There are two possible approaches to extend our service model in Degas with service composition. The user can explicitly request multiple services from the gateways. The user sets up the pipeline by linking the input sessions and output sessions of these services. The second approach is to hide service composition from the user. This is the approach we adopted. We let the gateway servicing the client decide how to decompose a deglet and how to distribute them across other gateways. We chose this approach to simplify the usage of the system and to avoid non-optimal configuration that can occur if a user did not set up the services and pipelines properly.

Under the new service model, the gateway servicing the client, called the *main gateway*, decomposes the deglet on the media streams into one *main deglet* and multiple sub-deglets. The sub-deglets are submitted to other gateways for execution. A gateway that runs sub-deglets is called a *helper gateway*. The main deglet remains on the main gateway and will be responsible for collecting input from the video sources and/or helper gateways, and performing final transformations on the output stream before sending the result to the client.

6.2 Assumptions and Constraints

We assume that a helper gateway can subscribe to any subset of sources in a multicast session, since a sub-deglet may only need certain streams as input. This is not possible currently as a receiver must receive data from all sources in the session the receiver subscribes to. However, this can be done in the future using Source-Specific Multicast [35] and Internet Group Management Protocol (IGMP) version 3 [12], which currently is an IETF draft.

There is a major disadvantage in sending a stream through multiple gateways – the latency between the sources and the receiver may increase because of the decoding and encoding operations that need to be performed at each gateway along a pipeline. We have shown in Section 4.4 that passing a stream through a gateway can introduce up to 30 ms of latency due to the decoding and re-encoding process. However, there is an important class of non-interactive applications where latency is less important, such as watching pre-recorded video streams. Furthermore, users can specify maximum latencies that they can tolerate in the system using preconditions (Section 4.2.2). We can constraint the system to split off a sub-deglet only if the total resulting end-to-end latency is smaller than the one specified by the user.

Another constraint of our system is that the service performed should not change its operations frequently. Otherwise, the involved deglet needs to be re-decomposed and re-assigned. An example of frequently changing deglets is one that filters input streams depending on who is the current speaker of a teleconferencing session (see Figure 4.5).

6.3 Research Goals

A research issue that arises is how the main gateway should decompose and distribute the deglet. There are several concerns. One concern is the resource requirements of the deglet. A sub-deglet should be assigned to a gateway that matches its resource requirements. For instance, we should assign a memory intensive sub-deglet to a gateway with sufficient memory. A main gateway with high CPU load should spawn off as many sub-deglets as possible.

A second concern is maximizing sharing between different services. A gateway can take sub-deglets that are already running on other gateways into consideration and try to share those services if they share the same sources and operations. A third concern is network bandwidth consumption. We should distribute the deglet so that the traffic between the gateways is as small as possible. A fourth concern is propagation delays. We should make sure that a gateway that is assigned to run a sub-deglet is not “out of the way”. It should be located relatively close to the path between the sender and the receiver. Making decisions based on multiple concerns that may conflict with each other is a complex problem. In this dissertation, we restrict ourselves to minimizing network bandwidth consumption.

We can express our goal as a graph problem: given a graph G representing gateways and links in the network, and a tree T representing the deglets, how to map the nodes in T onto nodes in G such that consumed network bandwidth is minimized? A polynomial solution to the problem can be found, but is not practical since the network environment is highly dynamic and a topology of all gateways in the network cannot be obtained easily. Therefore we opt for a decentralized approach, and decouple the problem into two independent subproblems: deglet decomposition and helper gateway assignment.

Hence, our research goals are to build a system that (1), automatically splits a high-level service requested by a user into sub-deglets, and (2), assign sub-deglets to gateways with the goal of reducing bandwidth consumption.

6.4 Computation Decomposition

In this section, we describe the algorithm we use to split a deglet into sub-deglets. As we decouple the problem of decomposing deglets and gateway assignments, we do not take network conditions or gateway topology into consideration. Instead, our algorithm tries to be optimistic and assume that a gateway is always available between a source and the main gateway to run the sub-deglet. Our algorithm uses the estimated size of compressed videos as a parameter to decide how to split a deglet, since we do not know what the actual size of the video will be at the time the decomposition occurs. We conservatively use the compression ratio of 50:1 for H.261 videos and 10:1 for MJPEG videos as estimates.

Our algorithm limits the number of gateways a stream can flow through to two. This greatly simplifies the decomposition algorithm. However, a stream may still flow through more than two gateways on its way to the receiver. A helper gateway can optionally act as a main gateway, and decompose the sub-deglet that is assigned to it using the same algorithm. These sub-sub-deglets can then be spawned off by the helper gateway to other gateways for execution.

6.4.1 Computation Model

We model the operations on a video stream as a tree. Leaf nodes in the tree correspond to the source of the video, and non-leaf nodes correspond to the operations performed on the video frames. An edge in the tree carries video frames and is associated with a weight value. The weight value corresponds to the data size of the video streams.

Formally, define a *computation tree* as a tree $G = (V_G, E_G)$ with a set of leaf nodes $V_{leaf} \subset V_G$ and a root node $V_{root} \in V_G$. Define a *weight function* on the edges as $w : E_G \to R^+$, and a *cut* (S, T) as a partition of V_G into two subsets S and T, such that $V_{leaf} \subseteq S$ and $V_{root} \in T$. We denote *root*(G) as the root of tree G and *cost*(E) as the sum of the weights of all edges in a set E. An edge (u, v) is said to *cross a cut* (S, T) if $u \in S$ and $v \in T$. The set of all edges that cross a cut (S, T) is called a *cut-set* for (S, T). A deglet is split into sub-deglets by removing edges that cross a cut. Each set of non-leaf nodes that still connect to each other after removing a cut-set corresponds to a sub-deglet. The sub-deglet that contains V_{root} will be the main deglet. See Figure 6.2 for an example.

As we assume that the sub-deglets will be assigned to different gateways for execution, the weight of the edges across the cut corresponds to the amount of

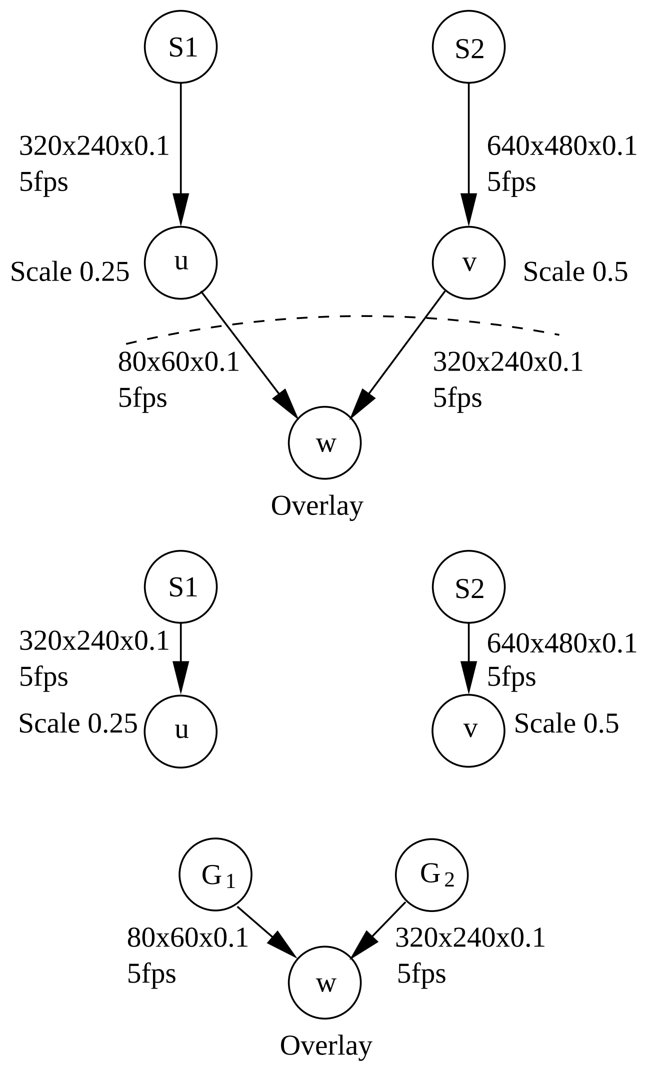


Figure 6.2: Example of a computation tree. Edges (u, w) and (v, w) divides the computation tree into three smaller computation trees, which correspond to the sub-deglets.

data to be sent across the network. Hence to find a cut that minimize network bandwidth consumption, we need to find a cut-set with minimum total weight, that is, we want to minimize $\sum w(u, v), u \in S$ and $v \in T$. Figure 6.3 shows our algorithm for finding the minimum cut-set (or *mincut*).

```
MinCut($G$)
1.  $E_{cut} \leftarrow \{\}$
2.  for each subtree $G_i$ of root($G$) do
3.      if $G_i$ is a single node then
4.          $E_{cut} \leftarrow E_{cut} \cup \{(root(G_i), root(G))\}$
5.      else
6.          $E'_{cut} \leftarrow$ MinCut($G_i$)
7.          if $cost(E'_{cut}) > w(root(G_i), root(G))$
8.              $E_{cut} \leftarrow E_{cut} \cup \{(root(G_i), root(G))\}$
9.          else
10.             $E_{cut} \leftarrow E_{cut} \cup E'_{cut}$
11. return $E_{cut}$
```

Figure 6.3: Algorithm for finding the minimum cut-set in a computation tree.

Our algorithm runs in time linear to the size of the computation tree, since it visits each edge exactly once. We provide an outline of the correctness proof below.

The proof is by induction on the depth of the computation tree. Consider the base case when the depth of the tree is one, that is, the tree consists of only the root node and the leaf nodes. In this case the algorithm will return $E_{cut} = E_G$.

This is the minimum cut-set since there is only one possible cut. Now assume that the algorithm works for trees with depth $< k$. Consider a tree of depth k. All subtrees must be of depth $< k$ and therefore MinCut will find the minimum cut-set of any subtree correctly. Now, consider the edge e that connects the root to a subtree G_i. A mincut of G must include either e or the mincut of G_i. Furthermore, if the weight of e is less than the cost of G_i's mincut, then e must be a member of the mincut of G. Otherwise we can replace the mincut of G_i with e and get a cut-set with lower cost and achieve a contradiction. Therefore, line 6 - 10 correctly find the edges that belongs to the mincut of G. By induction, we conclude that MinCut correctly finds the minimum cut-set of computation tree G.

We note that our algorithm works because we model the deglet as a tree, and is a special case of the general max flow/min cut problem. This is sufficient for most of the deglets we are interested in. A more generalized model of deglet, such as directed acyclic graph, would require a more complex algorithm. Such generalized computation models are required when sharing of sub-deglets is allowed among services, and are a subject of future research.

Once the main gateway uses MinCut to create a series of sub-deglets, it will need to locate other gateways to run these sub-deglets. We will describe our protocol for locating gateways in the next section.

6.5 Extension to AGLP

In this section, we describe how a main gateway locates and assigns sub-deglets to helper gateways. Our protocol extends AGLP. We call the extended version AGLP++. The process of locating helper gateways is very similar to the process of locating the main gateway by the client. However, instead of doing it in two stages (quick-start and adapt), we can do it in one since startup latency is no longer a concern.

We add a new phase, *Splitting Phase*, between the Quick-Start Phase and the Adapting Phase. The goal of the Splitting Phase is to split the deglet, and to request other gateways to help with the execution of the sub-deglets.

It is important that we defer the Adapting Phase until all the helper gateways are identified and initialized. The optimal locations to execute the main deglet and sub-deglets depend on each other, hence performing both the Adapting Phase and the Splitting Phase simultaneously will cause unnecessary migrations. The Adapting Phase is delayed by suppressing session information in the serve message. Without the session information, other gateways cannot evaluate and try to replace the main gateway.

The current gateway, G_0, initiates the Splitting Phase by multicasting a help-request message to all other gateways. The help-request message contains information about the sub-deglet to be executed on a gateway, including the input session and identities of the sources to the sub-deglet, the bandwidth of each input stream, and the distance of the main gateway from each source. This information is the same as the information sent with serve messages, except that only the subset of sources for the sub-deglet are sent.

A gateway that is available to help G_0, upon receiving a help-request message, evaluates itself to see if it is better than the main gateway for running the sub-deglet. The evaluation is carried out by calculating a score in a similar manner as the evaluation in the Adapting Phase.

Without loss of generality, let $S_0..S_{k'}$ be the subset of sources to the sub-deglet, and define b'_0 be the output bandwidth of the sub-deglet. We define U'_i as

$$U'_i = \sum_{j=0}^{k'} (b_j \times d_{i,j}) + b'_0 \times d_{i,0}$$

and score x'_i of a gateway G_i as

$$x'_i = U'_0 - U'_i$$

where $d_{i,i} = 0$. If a gateway gets a score larger than a threshold, the gateway waits for a certain amount of time before multicasting a help-offer message back to gateway G_0. A help-offer message is similar to a replace message. G_0 waits for a certain amount of time before picking a gateway with the highest score to run the sub-deglet. Let G_h be the one selected. G_0 then multicasts a help-accepted message, and hands off the sub-deglet to G_h. G_h subscribes to the sources, processes the video, and starts multicasting the processed video to G_0.

While G_0 is trying to find helper gateways to run the sub-deglets, G_0 continues to process the input streams using the un-decomposed deglet. Once G_0 knows that G_h is ready, G_0 reconfigures its deglet by removing the subtree that corresponds to the sub-deglet assigned to G_h. G_0 subscribes to the output session of G_h. G_h and G_0 periodically multicast a helping and helped-by message to each other to maintain the soft state relationship that G_h is helping G_0.

Just like the original AGLP, we need to be able to adapt to changing network conditions. G_h includes in its helping message the information about the session, and other gateways can evaluate themselves to see if they can replace G_h to help G_0. G_h can then hand off the sub-deglet to a better helper, and G_0 will switch its input from G_h to the new helper.

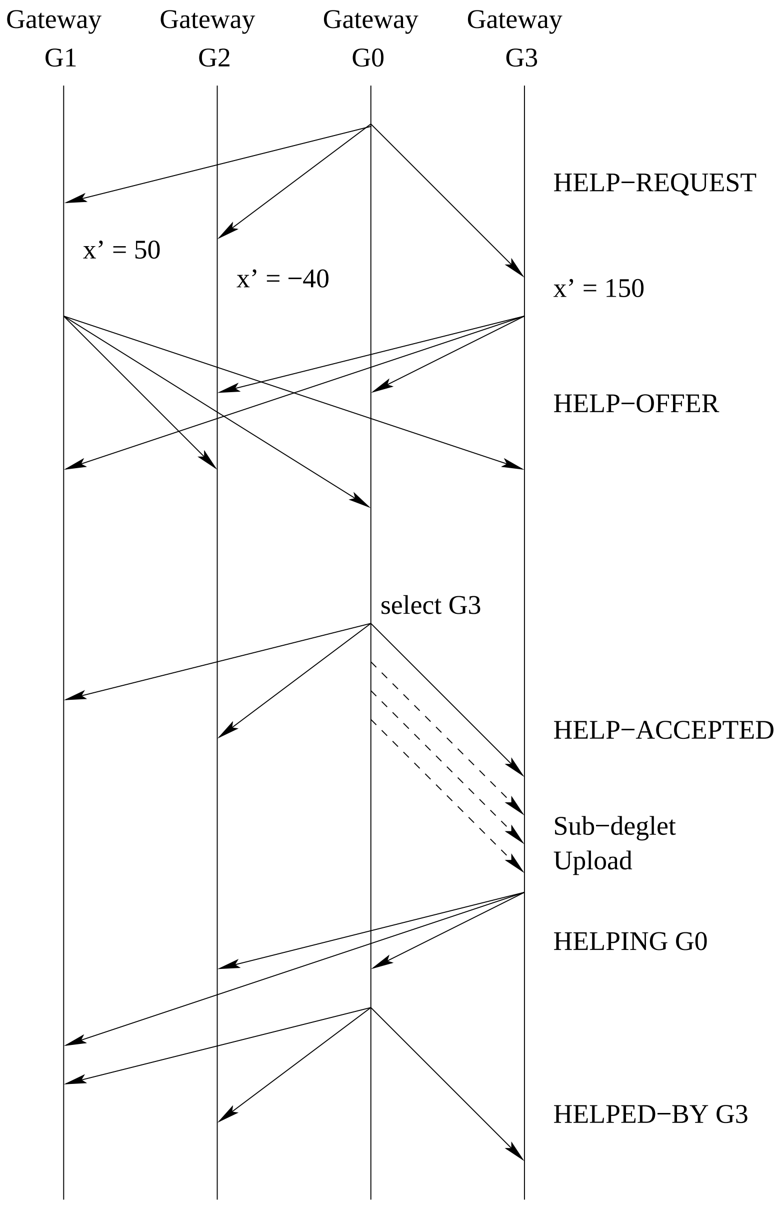


Figure 6.4: The Splitting Phase of AGLP++.

If no gateway replies to the help-request message from G_0, this implies that the best place to run the sub-deglet is on G_0, and the sub-deglet is not spawned. After a sub-deglet is assigned to a helper gateway, migrations might cause a sub-deglet and the main deglet to run on a same gateway again. These deglets can then be merged, by grafting the sub-deglet's computation tree back into the main deglet's computation tree.

The main gateway will initiate the Adapting Phase once it believes that it has reached a “stable” state, that is, it does not receive any more help-offer messages in T_{stable} seconds.

6.6 Performance of AGLP++

One particular issue that concerns us is how these changes to AGLP will affect the performance, in particular, how it will affect the number of migrations and time to reach the set of optimal gateways. We implemented the extension to AGLP in the ns2 simulator [6] and ran simulations on randomly generated 500-node networks using the gt-itm random topology generator [13]. We used a deglet similar to Figure 6.1(b) with three sources. The deglet is decomposed into three sub-deglets that resize the input streams, and a main deglet that merges the stream. In this section, we present our simulation results.

Figure 6.5 shows the number of migrations of the main gateway for AGLP and AGLP++. The result shows that the average number of migrations for AGLP++ is about the same as the original AGLP.

However, as we deferred the Adapting Phase until we assigned sub-deglets to helper gateways, the time it takes to migrate the main deglet to the optimal

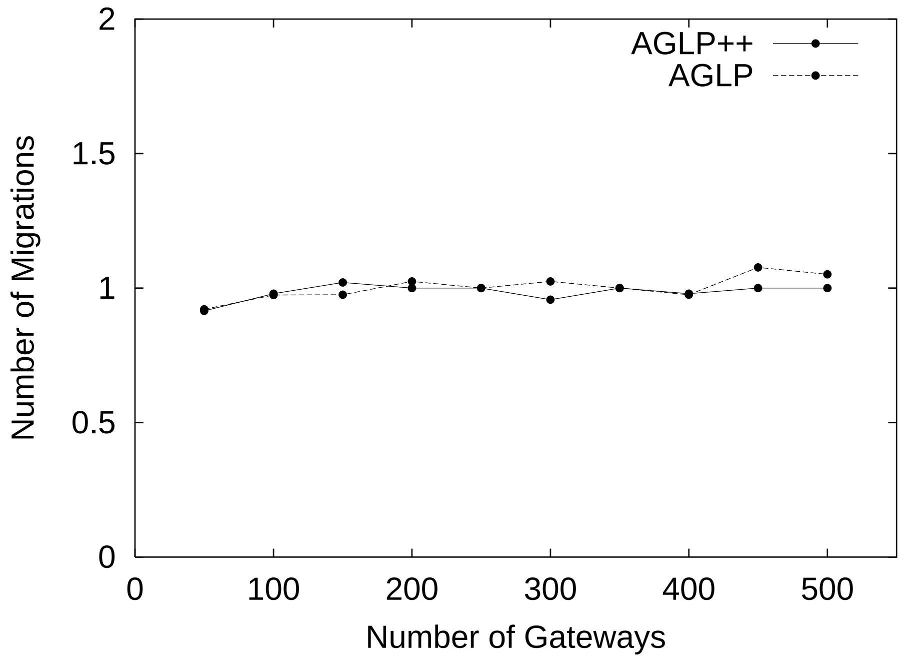


Figure 6.5: Number of migrations for different number of gateways.

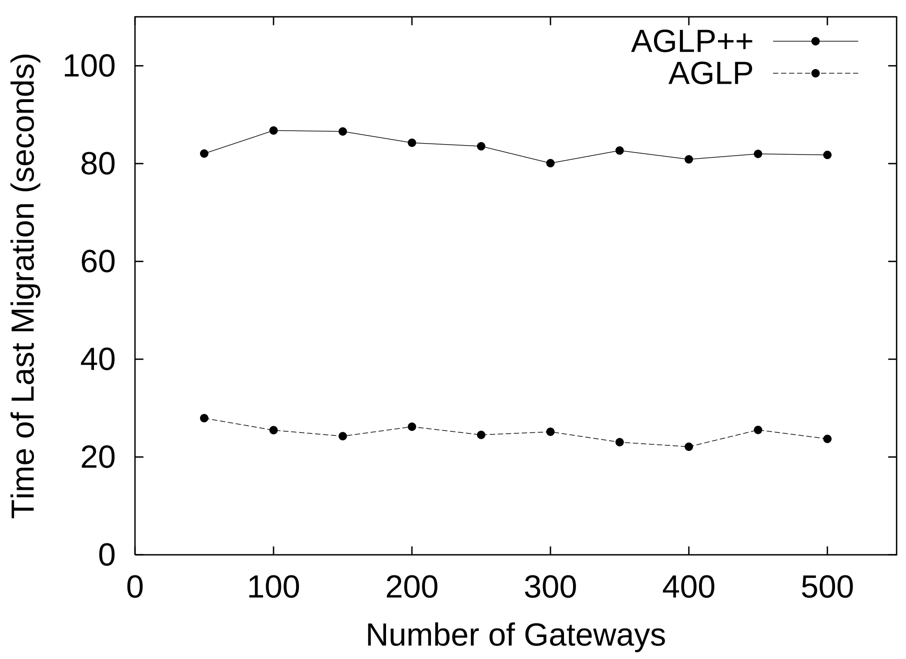


Figure 6.6: Time of last migration versus number of gateways.

gateway increases significantly. Figure 6.6 shows the time to reach the optimal gateway, plotted against the number of gateways. We used $T_{\text{stable}} = 30$ seconds in this simulation. The time to reach the optimal gateway increases by about 50 - 60 seconds.

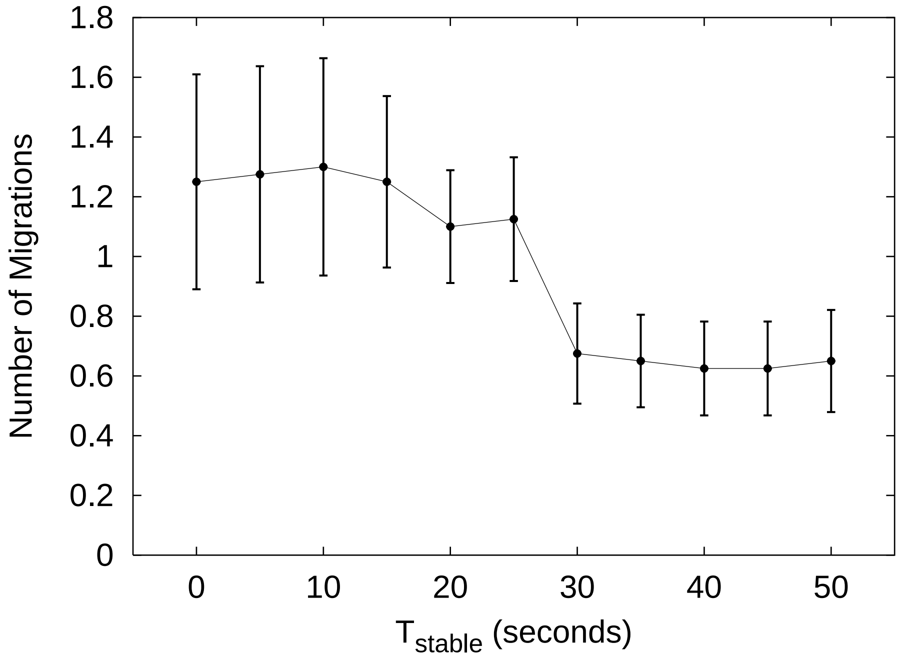


Figure 6.7: Average number of migrations for different values of T_{stable}, for 20 gateways, with 95% confidence interval.

T_{stable} is a parameter that we can tune to trade the number of migrations and the time to reach stability. A small T_{stable} causes optimal gateways to be found faster, but will cause the number of migrations to increase. The effect of this parameter is shown in Figure 6.7 and Figure 6.8.

An interesting observation from Figure 6.7 is that the average number of migrations drops significantly when T_{stable} is larger than 30 seconds. For values of T_{stable} less than 30 seconds, the Adaptive Phase starts before all sub-deglets are

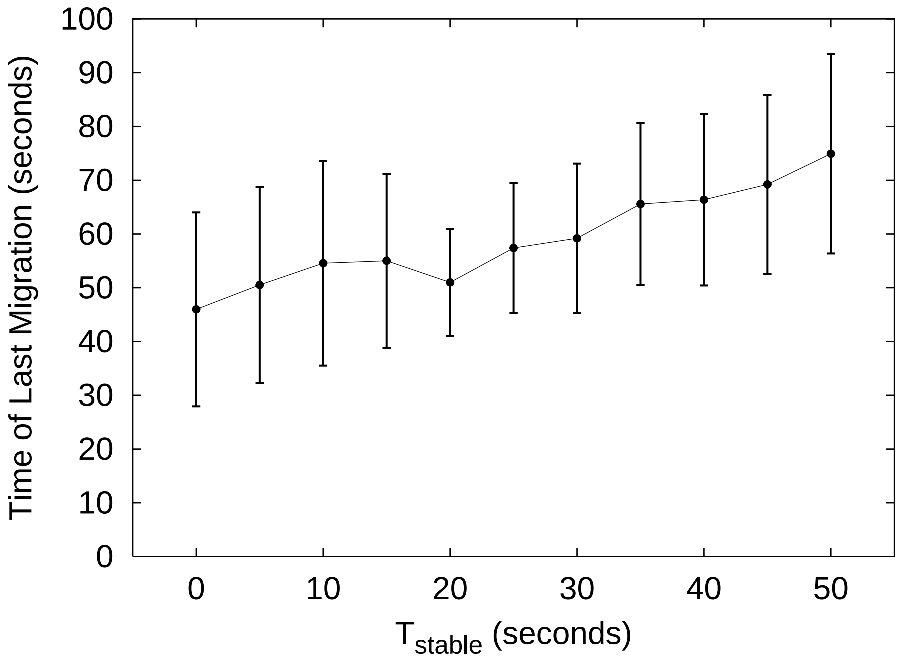


Figure 6.8: Average time of last migration for different values of T_{stable}, for 20 gateways, with 95% confidence interval.

assigned to helper gateways, and may unnecessarily migrate the main deglet to a gateway near the sources.

Figure 6.9 shows the average number of times helper gateways start executing sub-deglets. As the main gateway in our simulation requested help for three sub-deglets, a value of 3 indicates that all sub-deglets are assigned to their respective optimal helper gateway the first time. A value larger than 3 implies some migrations of sub-deglets. This graph shows that the number of migrations per sub-deglet is not much larger than 1, indicating that we are able to locate good helper gateways to run the sub-deglets with one migration most of the time.

Figure 6.10 shows the average number of help-offer messages that the main gateway receives per number of sub-deglets. Just as the replace messages during the Adapting Phase, we use multicast damping to avoid feedback implosion. The

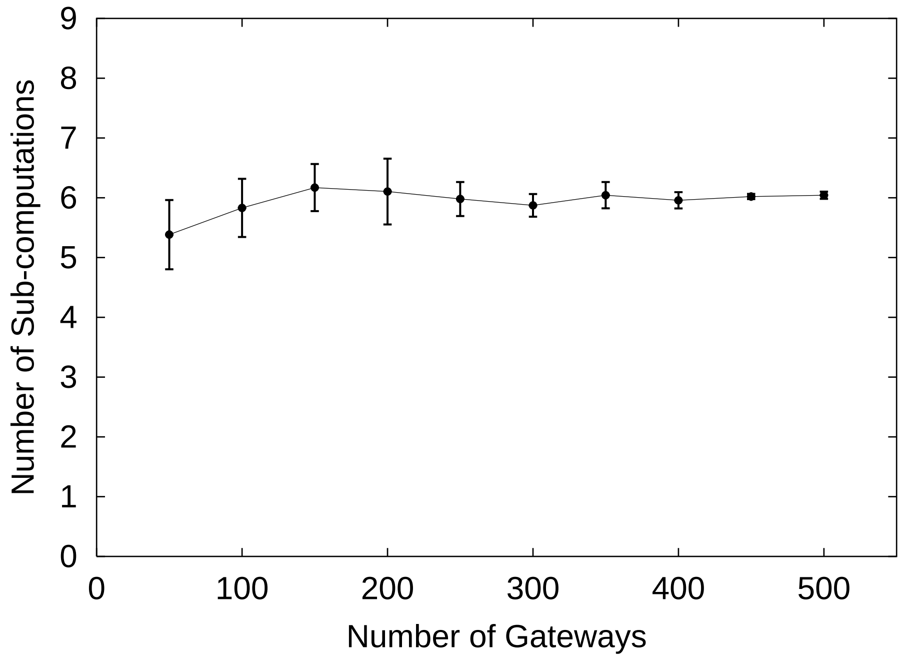


Figure 6.9: Average number of sub-deglet executions, with 95% confidence interval.

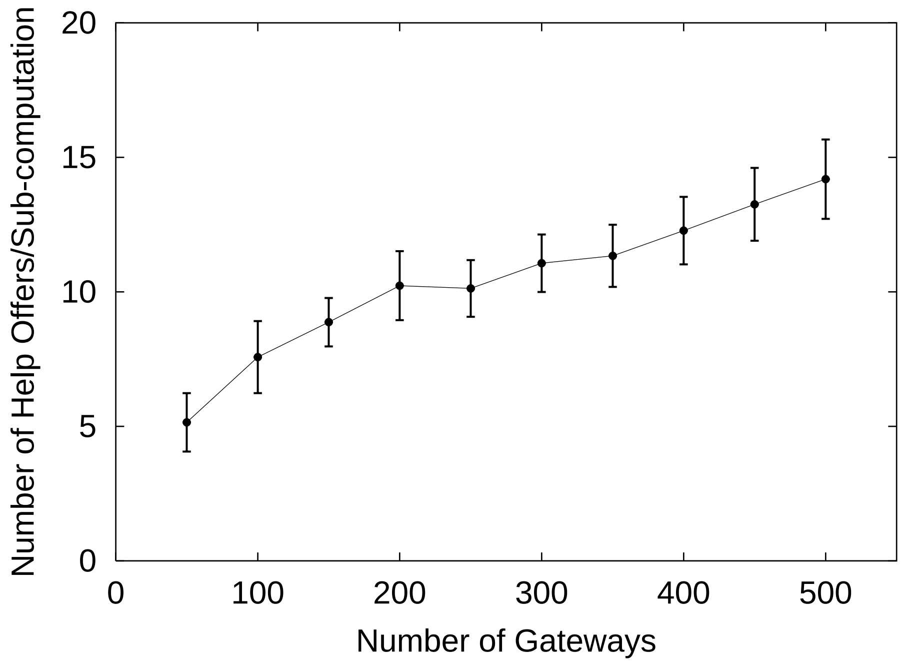


Figure 6.10: Average number of help messages, with 95% confidence interval.

graph shows that this number increases very slowly as the number of gateways goes from 50 to 500. A caveat here is that this graph shows the number of messages per sub-deglet. The number of help-offer messages increases linearly with the number of sub-deglets, and hence does not scale well. One solution to this problem is to request help for the sub-deglets sequentially, instead of simultaneously. This can spread the help-offer messages over a period of time and avoid implosion. However, this can increase the time to reach optimal configuration significantly.

In summary, our simulation results show that AGLP can be extended to locate media gateways for running composable services, while still maintaining a low number of migrations, and a low number of messages at the expense of more time to reach optimal gateways.

6.7 Effects on QoS

In this section, we present experimental results of our system to study the effects of distributed media transformation on the quality of video received by the receiver. The deglet that we used in these experiments is shown in Figure 6.11. This deglet merges two high quality MJPEG streams into a low-quality, 5 fps H.261 stream.

We first ran the deglet on a single gateway. To simulate the situation of an overloaded gateway, we chose a slow machine, a Sun SPARCstation-5 with 32 MB of RAM (called host A) as our gateway. In the second experiment, we split the same deglet into three operations u, v and w, and distributed the operations onto three gateways. We used two Sun Ultra-80 computing servers as helper gateways to run operations u and v, and ran operation w on host A as the main gateway.

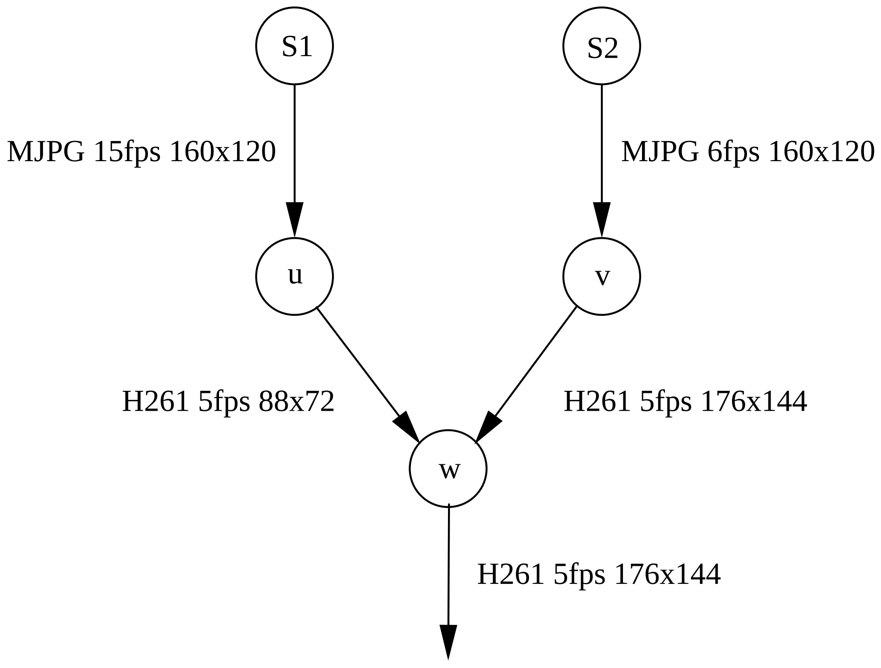


Figure 6.11: A picture-in-picture deglet used in experiments.

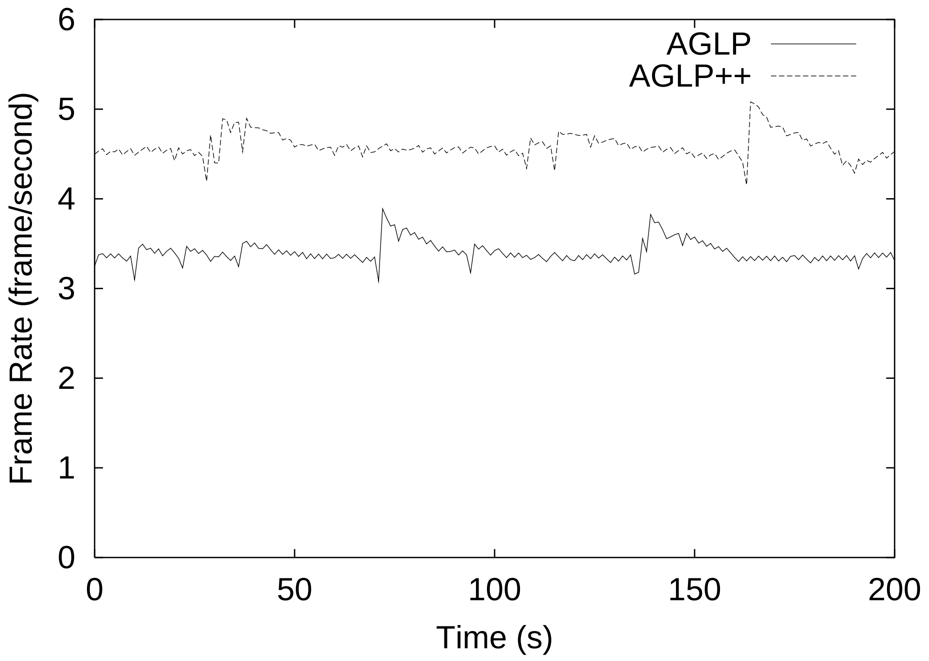


Figure 6.12: Frame rate versus time.

Figure 6.12 shows the number of frames per second received by the client for both experiments. In the case where a single gateway is used, the receiver only receives about 3.3 fps. The CPU load on host A is about 85%. By assigning some computing intensive sub-deglets to helper gateways, we are able to lower the CPU load on host A to about 25%, and improve the frame-rate close to 5 fps. Figure 6.13 shows the corresponding data rate received in the experiments.

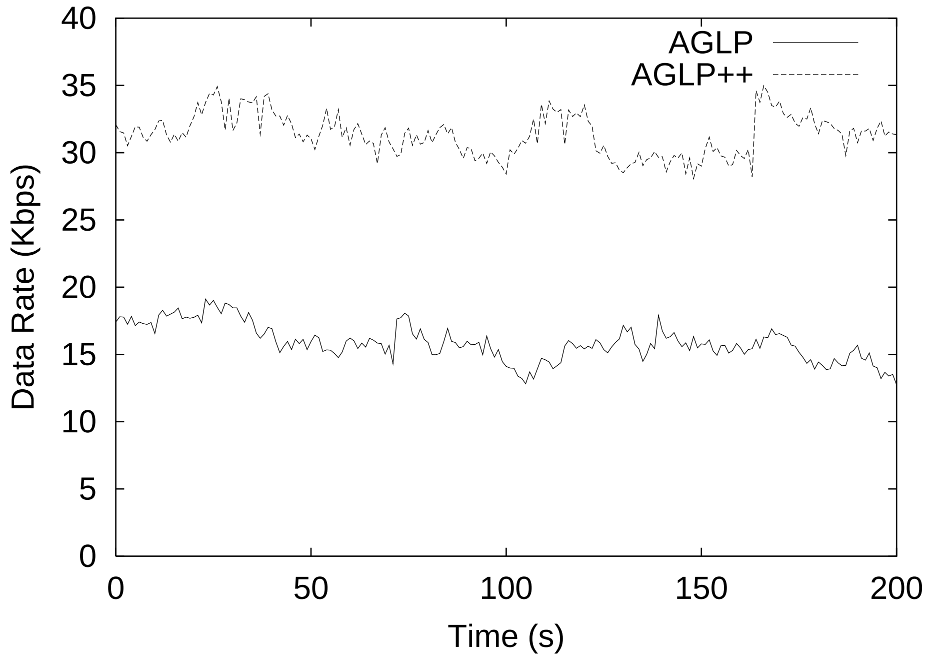


Figure 6.13: Data rate versus time.

We measured the period between rendering of frames for both experiments. The results are shown in Figure 6.14 and Figure 6.15. Our result shows that we are able to reduce jitter significantly by using multiple gateways.

A surprising result from these experiments is our measurement of end-to-end delays. We expected the end-to-end delay for running a deglet on multiple gateways to be larger than running it on a single gateway. However, we found that the end-to-end delay is about 600ms higher when we ran the deglet on a single gateway.

This is because the frame processing time on host A is much larger than the time spent in the extra decoding/encoding process when the stream is passing through a second gateway.

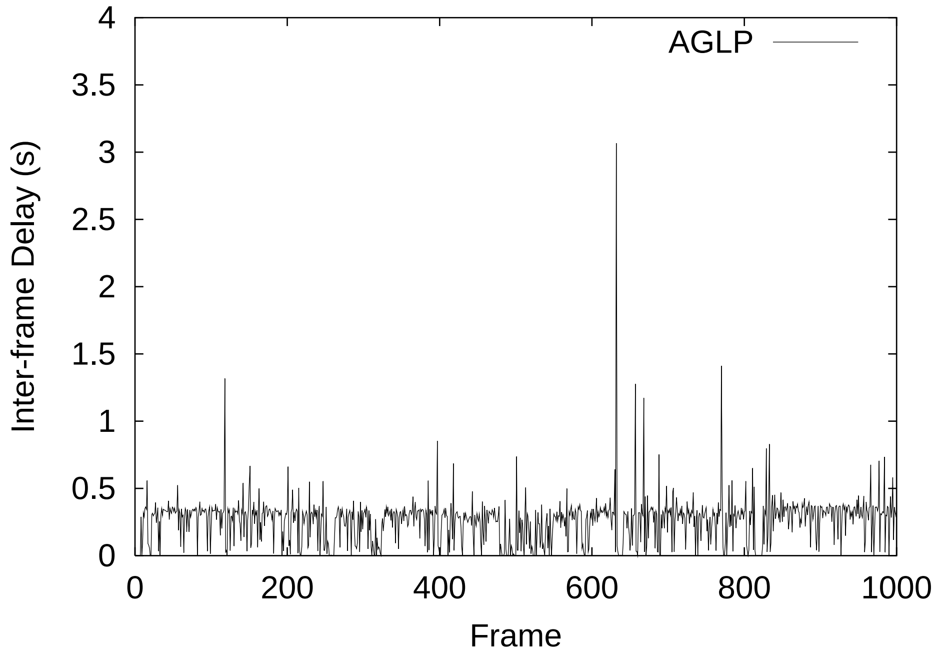


Figure 6.14: Inter-frame rendering delay for a single gateway.

The results of our experiments are very encouraging. They validate our believe that by distributing media transformation over multiple gateways, we are able to improve throughput. In cases when a single gateway becomes bottleneck, helper gateways can help reduce jitter and end-to-end delay, thus improving the quality of the video streams received by the user.

6.8 Conclusion

In this chapter, we described mechanisms to distribute a deglet over multiple gateways for execution. The deglet is first decomposed into sub-deglets, and is assigned to helper gateways using a modified version of AGLP. Our simulations show that

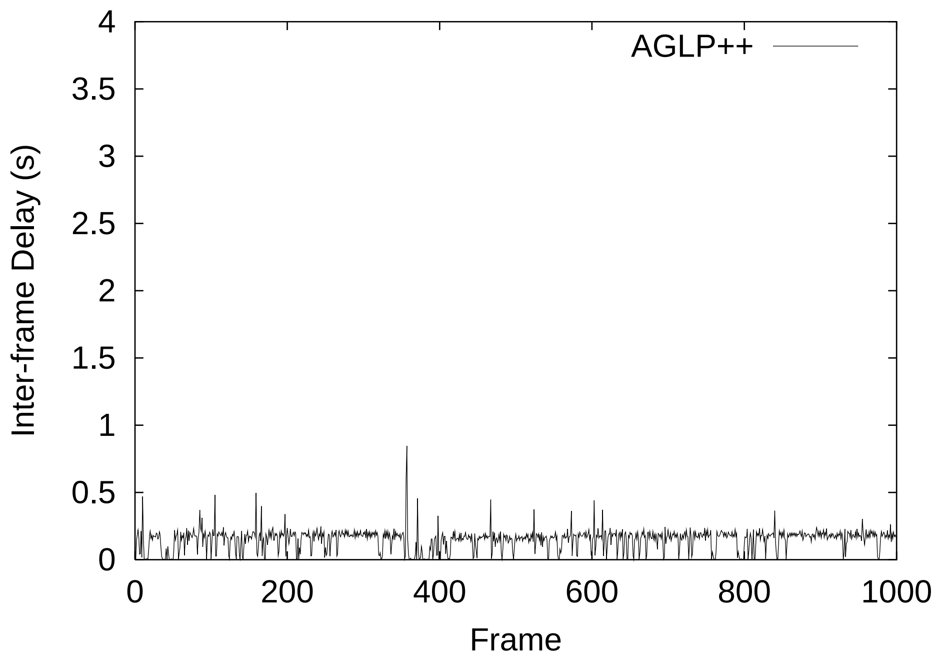


Figure 6.15: Inter-frame rendering delay for multiple gateways.

our extension does not affect AGLP significantly, and experiments show that distributing deglets can improve the QoS of the resulting stream.

Our work in distributed deglet execution is only preliminary. There are many problems that remain to be solved. We outline these problems in the next chapter.

Chapter 7

Conclusion and Future Work

Our work in Dali, Degas, AGLP and AGLP++ has contributed toward the realization of a distributed programmable media gateway service on a wide-area network. However, there are still many issues that need to be addressed. We present these issues in this chapter.

7.1 Degas Architecture

We built Degas as a prototype of a programmable, application-level media gateway. Our prototype serves as a framework where many research issues about the design of programmable media gateways can be explored.

7.1.1 Deglet Optimization

Currently, we only perform per-operation optimization when we execute a deglet. We translate a frame operation to the optimal sequence of corresponding Dali operations. However, we did not perform optimization over a sequence of frame operations. Instead, we rely on the programmer to write an efficient deglet.

To provide a complete solution, a compiler is needed. The compiler takes in a high-level program that describes the operations on the video streams, optimizes the program, and generates low-level Dali calls. Optimization strategies similar to the ones in Rivl [73] can be used. Rivl rearranges the operations to run those that reduce data size first. For example, cropping of a frame is performed before scaling. It also calculates the regions in the inputs that are needed in the output, and only processes those regions. Hence, if we are overlaying video frame A onto a video frame B, the area on B that is covered by A need not be computed. Our decomposition algorithm described in Section 6.4 should be integrated with this compiler.

7.1.2 Multiple Receivers

In the current design of Degas, we assume that clients are independent. In practice, multiple clients may be interested in sharing the same services from Degas. We can easily allow multiple clients to receive post-processed streams from gateways. Since the post-processed video stream is multicast to an output session, a host that is interested in a post-processed stream can tune in to the session to receive the stream. This can be done as follows.

We can augment the Session Description Protocol [32] to include information about services currently provided by the gateways. A user can view the list of services available using a GUI front end, and join any desirable session. The corresponding host will periodically announce served-by messages onto the common multicast channel. The served-by messages from multiple receivers can be consolidated by using multicast damping: if a receiver R receives a served-by message, then R reschedules the announcement of its own served-by message. This reduces the total number of served-by message sent. If the original client C that initiated the service quits, the gateway will continue serving the other receivers as it is still receiving served-by messages.

Two problems arise. First, what if the gateway servicing the receivers fails after C quits? The other receivers do not have access to the original deglet submitted by C, and therefore cannot restart the service. One possible solution to the first problem is to have each receiver download the program from the gateway (if C permits it) as they join the output session. Another solution is to let the gateway periodically multicast the program onto a separate channel.

The second problem concerns the calculation of scores during evaluation. Since the output from the gateway is now multicast to multiple receivers, how can we characterize the bandwidth consumption of the output stream? We can estimate the propagation delay from the gateway to all receivers by using receiver report RTCP packets, but since resources are shared in the multicast tree, we cannot simply sum the products of the bandwidth and the distances.

7.1.3 Resource Management

Resource management is a crucial component in a user-extensible media gateway. First, we need to prevent users from submitting malicious resource-hogging deglets, such as one that runs in infinite loops or unnecessarily allocates huge amounts of memory. Secondly, we need to ensure that resources are distributed fairly among users, or at least the deglets in a gateway. An interesting issue is when and how to revoke resources from a running deglet. 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.

There are a few common strategies we can employ to manage resources. Resources can be reserved at the beginning of the execution of a deglet. The resource manager then has to police the resource usage of each deglet, and notify the deglet to reduce resource consumption when it exceeds the reservation. Reservation of existing deglets may be involuntarily reduced when new deglets start, in order to ensure fairness. A problem with reservation-based schemes is that they often lead to over-reservations and waste of resources. Another common approach is market-based resource allocation, where each deglet starts out with some "money" which it can use to bid for resources. The price for the resources increases when it is in demand (for example, when the number of deglets running increases). A deglet has to relinquish some of its resources when it runs out of money.

In both schemes described above, resources can be dynamically revoked (when a deglet runs out of money, or when it uses more than what it reserved). We can implement a scheme similar to [54] in Degas. Degas can force a deglet to reduce the quality of the output stream when resources need to be revoked. This includes reducing the size (resolution) of the output video and converting the output video to gray-scale. These reductions can free up some of the resources, including CPU, memory and network. If the deglet still consumes too much resources, the gateway can force the deglet to be migrated to another gateway. The user can specify a preference to the action performed when resources are revoked. A drawback of this approach is that constantly changing the quality of video might irritate the users.

7.1.4 Security

Security is another common concern in extensible architectures. However, we believe that these concerns can be easily addressed in Degas. As in Safe-Tcl [56], 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 [11]). Gateways would have access to a user database that describes access control policies.

7.2 AGLP

AGLP minimizes bandwidth consumption by executing deglets at strategic locations in the network. AGLP currently based its decision solely on the measured propagation time between the hosts. Other possibilities exist.

7.2.1 Exploiting the Multicast Tree

AGLP does not take the multicast tree into consideration when it locates gateways for services. This works well enough when we have a single, remote host that requested a gateway service. However, if there are other near-by hosts participating in the same session, this could affect the optimal location of the gateway. We illustrate this in an example shown in Figure 7.1.

Figure 7.1 (a) shows a multicast tree with node a as the source. Both node b and c are media gateways. Figure 7.1 (b) shows the scenario where node e requests a service to be performed on the video stream from node a. Supposing that the service e requested is one that reduces the bandwidth, AGLP will assign the service to a gateway that is near node a, in this case, node b. The output stream is transmitted through path $b-c-e$ to reach e.

Now suppose that node d decides to receive the stream directly from node a. (See Figure 7.1 (c)). The original video stream now flows through the path $b-c-d$

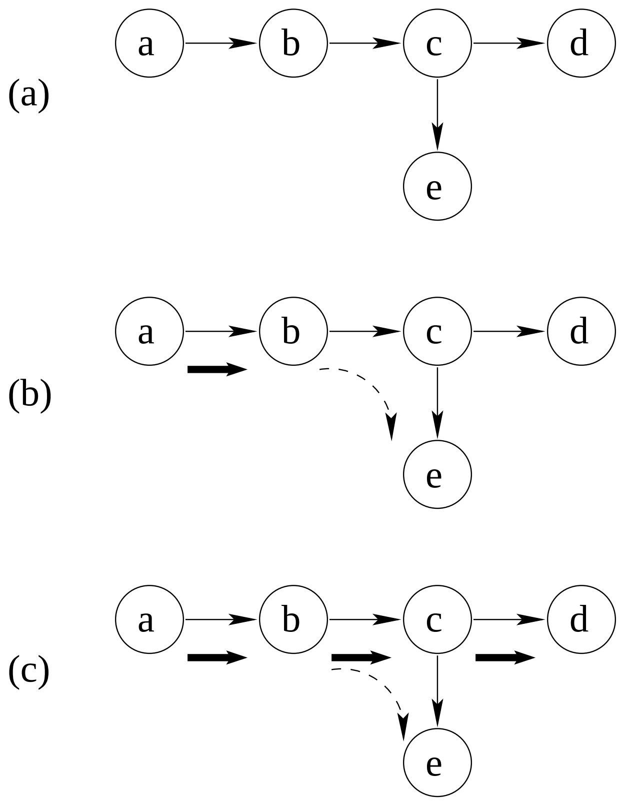


Figure 7.1: AGLP and multicast tree

to reach the destination d. In this case, choosing node b as the servicing gateway is not a good choice, since both the original and processed stream flow through $b - c$. A better gateway is node c.

Discovering the topology of the multicast tree so that it can be coorperated into AGLP is not trivial. A possible solution is described in [43]. Each participating host performs a trace of the multicast path periodically using the multicast version of `traceroute` called `mtrace`. They then share the path information with each other so that a tree can be constructed. We plan to look into how to do this in AGLP in a scalable manner.

7.2.2 Optimization Metrics

AGLP picks a gateway that minimizes bandwidth consumption. There are many other metrics we could consider when deciding which gateway should run a deglet.

Resource Availability. Available resources, such as the amount of available CPU, memory and network bandwidth should be considered when assigning deglets to gateways. Availability of special hardware, such as decoding and encoding hardware or media processing chips should be considered as well.

Load Balancing. There are cases when it is desirable to balance the load of multiple gateways. Hence AGLP should look at the loads on multiple gateways and assign the deglet such that the loads are as balanced as possible.

However, it is not clear how to integrate these different and often contradicting metrics into one variable in order to decide which gateway is more suitable to service a particular client.

7.3 Distributed Media Processing

7.3.1 Distributing Dynamic Deglets

One of the constraints on running composable services efficiently is that the services performed must be relatively static, that is, the computation does not change frequently depending on the inputs. We find that this limits the application of composable services in Degas, as many interesting applications of our gateway involve deglets that change their operations dynamically. For instance, in a video chat-room application where participants join and leave often, or in a distance learning application where we switch the output video streams between the lecturer and the audience based on who is currently talking. Even for a deglet that is static, the resource manager might still force the deglet to change its resolution or color space when its resources are revoked. Due to the importance of this class of deglets, we plan to relax this constraint in the future.

A challenge for distributing frequently changing deglets is to adapt to changes efficiently and in a responsive manner. The computation might need to be re-decomposed and redistributed to the helper gateways. The helper gateways need to be notified of the changes. We believe some kind of incremental algorithm can be used to adapt to the changes efficiently.

7.3.2 Fairness Among Users

The resource management policy in the gateway maintains fairness among different running deglets. However, one user might still use more resources than others. This can happen when a user runs multiple deglets, or when a deglet is distributed among different gateways. Hence, fairness among users should be maintained as well. We want the total resources consumed by a user over all gateways to be as fair as possible.

Ensuring fairness among users is difficult since it requires a global view of all gateways. Furthermore, finding the optimal assignment that maximizes fairness is a generalized case of non-uniform load balancing problem and is NP-complete [42]. We are looking at different strategies to achieve approximate fairness in a decentralized and scalable manner. A possible strategy that we have tried is as follows. Each gateway periodically notifies nearby gateways about the deglets that consumed the most and the least resources relative to their fair share. The gateways exchange these deglets if the exchanges will result in an improvement in overall fairness. Although we are able to increase overall fairness, there is room for improvement in our results.

7.3.3 Sharing Sub-computations

A particularly interesting idea that surfaced from distributing a deglet over multiple gateways is the notion of sub-computation sharing – sub-computations from different deglets that perform the same operations on the same video streams can be shared, hence only one copy need to run. Opportunity for sharing sub-computations seems to be rare in small sessions. However, the probability of sharing increases as the session size increases. For a large session such as live video broadcast of a concert, common operations such as transcoding into a popular format should be shareable.

A problem that needs to be solved is how such shareable sub-computations can be identified. A gateway needs to know what nearby gateways are currently running. Such information can be periodically multicast along with AGLP's serve and helping messages. Although determining whether two sub-computations are equivalent is unsolvable in the general case, we can restrict ourselves to simple sequences of operations, such as scaling by half and transcoding. By comparing the operations, identical sub-computations can be identified and shared.

7.4 Availability of Software

In this dissertation, we propose a solution to the problem of efficient media delivery over heterogenous networks. We developed several components of the media gateways. First, we built Dali, a high-perfomance software library that forms the execution engine for our gateways. Next, we built Degas, a prototype of our media gateways system, which serves as a framework for exploring research issues in programmable media gateways. Finally, we simulated and implemented AGLP and AGLP++, two protocols for locating media gateways in the network to perform transformations. All of these components are publicly available through our web sites. Dali can be downloaded from `http://www.cs.cornell.edu/dali`, while Degas and the media gateway location protocols are available at `http://www.cs.cornell.edu/degas`.

Bibliography

[1] S. Acharya and B. C. Smith. Compressed domain transcoding of MPEG. In *Proceedings of the International Conference on Multimedia Computing and System (ICMCS)*, pages 295–304, Austin, Texas, June 1998.

[2] S. Agamanolis. Isis - A multilevel scripting environment for responsive multimedia. Software Online, `http://isis.www.media.mit.edu/projects/isis/`.

[3] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, May 1998.

[4] E. Amir, S. McCanne, and Z. Hui. An application level video gateway. In *Proceedings of 3rd ACM International Multimedia Conference and Exhibition*, pages 255–266, San Francisco, CA, November 1995.

[5] E. Amir, S. McCanne, and R. Katz. An Active Service framework and its application to real-time multimedia transcoding. In *Proceedings of ACM SIGCOMM*, pages 178–189, Vancouver, Canada, August 1998.

[6] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999.

[7] G. Ballintijn and M. van Steen. Characterizing Internet performance to support wide-area application development. *Operating System Review*, 34(4):41–47, August 2000.

[8] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An active transcoding proxy to support mobile web access. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 118–123, West Lafayette, Indiana, October 1998.

[9] T. W. Bickmore and B. N. Schilit. Digestor: Device-independent access to the World Wide Web. *Computer Networks and ISDN Systems*, 29(8–13):1075–1082, 1997.

[10] T. Boutell. A graphics library for fast GIF creation. Software Online, `http://www.boutell.com/gd/`.

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

[12] B. Cain, S. Deering, B. Fenner, I. Kouvelas, and A. Thyagarajan. Internet Group Management Protocol, version 3 `http://www.ietf.org/internet-drafts/draft-ietf-idmr-igmp-v3-05.txt`, November 2000.

[13] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 36(6):160–163, June 1997.

[14] J. Carter, W. Hsieh, M. Swanson, L. Zhang, A. Davis, M. Parker, L. Schaelicke, L. Stoller, and T. Tateyama. *Memory system support for irregular applications*, pages 17–26. Springer-Verlag, Lecture Notes in Computer Science 1511, 1998.

[15] D. D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of ACM SIGCOMM*, pages 106–114, Stanford, CA, aug 1988.

[16] K. G. Coffman and A. M. Odlyzko. *Internet growth: Is there a Moore's Law for data traffic?* Kluwer, Norwell, MA, 2001.

[17] Intel Corporation. *MMX Technology Programmers Reference Manual*, 1997.

[18] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[19] B. Davidsen. NETPBM graphics package. Software Online, `ftp://ftp.cs.ubc.ca/ftp/archive/netpbm/netpbm-1mar1994.tar.gz`, March 1993.

[20] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989.

[21] R. Droms. RFC 2131: Dynamic host configuration protocol, March 1997.

[22] H. Eriksson. MBone: the multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.

[23] W. Fenner. RFC 2236: Internet Group Management Protocol, version 2, November 1997.

[24] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[25] A. Fox, S. D. Gribble, Y. Chawathe, and E. Brewer. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, St. Malo, France, October 1997.

[26] R. Frederick. Experiences with real-time software video compression. In *Proceedings of The 6th International Workshop on Packet Video*, Portland, Oregon, September 1994.

[27] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, adaptive network services infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Franscisco, California, March 2001.

[28] D. Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, 34, 4:46–58, 1991.

[29] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Computer Networks (Special Issue on Pervasive Computing)*, 35(4):473–497, March 2001.

[30] Independent JPEG Group. JPEG library, release 6b. Software Online, `ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz`, March 1998.

[31] M. Handley. The sdr session directory tool. Software Online, `ftp://cs.ucl.ac.uk/mice/sdr/`, November 1995.

[32] M. Handley and V. Jacobson. RFC 2327: SDP: Session description protocol, April 1998.

[33] M. Handley, C. Perkins, and E. Whelan. RFC 2974: Session announcement protocol, October 2000.

[34] V. Hardman. rat - Robust Audio Tool. Software Online, `http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/`.

[35] H. Holbrook and B. Cain. Source-specific multicast for IP, `http://www.ietf.org/internet-drafts/draft-holbrook-ssm-arch-02.txt`, March 2001.

[36] L. Holden. ooMPEG: Object-oriented MPEG decoder. Software Online, `http://www.cs.brown.edu/software/ooMPEG/`.

[37] InternetNews.com. GEO, Samsung unveil MPEG4 video cell phone. InternetNews, `http://www.internetnews.com/prod-news/article/0,,9_513381,00.html`, November 2000.

[38] V. Jacobson. Multimedia conferencing on the Internet. ACM SIGCOMM 1994 Tutorial.

[39] V. Jacobson and S. McCanne. vat - LBNL audio conferencing tool. Software Online, `http://www-nrg.ee.lbl.gov/vat`.

[40] V. Jacobson and S. McCanne. wb - LBNL white board. Software Online, `ftp://www-nrg.ee.lbl.gov/wb/`.

[41] D. McDonough Jr. Firepad brings streaming video to Palm PDA. Wireless Newsfactor, `http://www.wirelessnewsfactor.com/perl/story/6853.html`, January 2001.

[42] J. M. Kleinberg, Y. Rabani, and E. Tardos. Fairness in routing and load balancing. In *IEEE Symposium on Foundations of Computer Science*, pages 568–578, 1999.

[43] B. N. Levine, S. Paul, and J. J. Garcia-Luna-Aceves. Organizing multicast receivers deterministically by packet-loss correlation. In *Proceedings of ACM Multimedia 1998*, pages 201–210, Bristol, England, 1998.

[44] C. Lindblad, D. Wetherall, and D. L. Tennenhouse. The VuSystem: A programming system for visual processing of digital video. In *ACM Multimedia*, pages 307–314, San Francisco, CA, 1994.

[45] S. Mann. A GNU/Linux wristwatch videophone. Linux Journal, `http://www2.linuxjournal.com/lj-issues/issue75/3993.html`, June 2000.

[46] J. Matthews, P. A. Gloor, and F. Makedon. VideoScheme: A programmable video editing system for automation and media recognition. Technical Report TR93-187, Dartmouth College, Computer Science, January 1993.

[47] K. Mayer-Patel and L. Rowe. Exploiting temporal parallelism for software-only video effects. In *Proceedings of the 6th ACM International Conference on Multimedia*, pages 161–169, Bristol, England, September 1998.

[48] K. Mayer-Patel and L. Rowe. Exploiting spatial parallelism for software-only video effects. In *Proceedings of Multimedia Computing and Networking 1999, Proceedings of the SPIE, vol. 3654*, pages 252–263, San Jose, California, January 1999.

[49] 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. C. Smith. Toward a common infrastructure

for multimedia-networking middleware. In *Proceedings of 7th. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'97)*, pages 39–49, St. Louis, Missouri, May 1997.

[50] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *Proceedings of 3rd ACM Intl. Multimedia Conf. and Exhibition*, pages 511–522, San Francisco, CA, November 1995.

[51] Sun Microsystems. Jini technology specification. `http://www.sun.com/jini/specs/`.

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

[53] J. Nonnenmacher and E. W. Biersack. Scalable feedback for large groups. *IEEE/ACM Transactions on Networking 1999*, 7(3):375–386, June 1999.

[54] G. J. Nutt, S. Brandt, A. Griff, S. Siewert, M. Humphrey, and T. Berk. Dynamically negotiated resource management for data intensive application suites. *Knowledge and Data Engineering*, 12(1):78–95, 2000.

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

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

[57] 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.

[58] K. Patel, B. C. Smith, and L. Rowe. Performance of a software MPEG video decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, Anaheim, CA, August 1993.

[59] N. Patel and I. Sethi. Compressed video processing for cut detection. In *IEEE Proceedings: Vision, Image and Signal Processing (Vol. 143)*, pages 315–323, October 1996.

[60] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, NY, 1993.

[61] D. Plonka. Internet traffic flow size analysis. `http://net.doit.wisc.edu/data/flow/size/`, 2000.

[62] E. J. Posnak, R. G. Lavender, and H. M. Vin. An adaptive framework for developing multimedia software components. *Communications of the ACM*, 40(10):43–47, October 1997.

[63] H. K. Pung and N. Bajrach. A programmable ATM multicast service with congestions control. *IEICE Transaction in Communication*, E83-B(2):253–263, February 2000.

[64] R. S. Ramanujan and K. J. Thurber. An active network based design of a QoS adaptive video multicast service. In *Proceedings of the 1998 World Conference on Systems, Cybernetics and Informatics*, pages 643–650, Orlando, Florida, July 1998.

[65] A. Schuett, S. Raman, Y. Chawathe, and R. Katz. A soft state protocol for accessing multimedia archives. In *Proceedings of The 8th. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSS-DAV'98)*, pages 29–40, Cambridge, UK, July 1998.

[66] H. Schulzrinne. Voice communication across the Internet: a network voice terminal. Technical Report 92–50, University of Massachusetts at Amherst, Amherst, Massachusetts, July 1992.

[67] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, January 1996.

[68] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared passive network performance discovery. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[69] B. Shen and I. Sethi. Convolution-based edge detection for image/video in block DCT domain. *Journal of Visual Communication and Image Representation*, 7(4):411–423, 1996.

[70] M. Singer. Get streaming video in your Palm. InternetNews, `http://www.internetnews.com/streaming-news/article/0,,8161_750201,00.html`, April 2001.

[71] B. C. Smith. *Implementation Techniques for Continuous Media Systems and Applications.* PhD thesis, Department of Computer Science, University of California, Berkeley, December 1994.

[72] B. C. Smith and L. A. Rowe. Compressed domain processing of JPEG-encoded images. *Real-Time Imaging*, 2(1):3–17, February 1996.

[73] J. Swartz and B. C. Smith. Rivl: a resolution independent video language. In *Proceedings of the Tcl/TK Workshop*, pages 235–242, 1995.

[74] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[75] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *International Conference on Distributed Computing Systems*, pages 234–243, 1999.

[76] T. L. Tung. MediaBoard: A shared whiteboard application for the MBone. Master's thesis, Computer Science Division (EECS), University of California, Berkeley, CA, January 1998.

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

[78] T. Turletti and J. Bolot. Issues with multicast video distribution in heterogeneous packet networks. In *Proceedings of The 6th International Workshop on Packet Video*, pages F3.1–3.4, Portland, Oregon, September 1994.

[79] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. RFC 2165: Service location protocol, June 1997.

[80] D. Wetherall. OTcl tutorial. `http://www.isi.edu/nsnam/otcl/doc/tutorial.html`.

[81] M. Williams. LG launches Internet refrigerator. IDG News Service, `http://www.idg.net/idgns/2000/06/23/LGLaunchesInternetRefrigerator.shtml`, June 2000.

[82] A. Wolman, G. Veolker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS)*, October 1999.

[83] T. Wong, K. Mayer-Patel, D. Simpson, and L. Rowe. Software-only video production switcher for the Internet MBone. In *Proceedings of SPIE Multimedia Computing and Networking*, pages 28–39, San Jose, CA, 1998.

[84] D. Xu, K. Nahrstedt, and D. Wichadakul. MeGaDiP: a wide-area media gateway discovery protocol. In *Proceedings of IEEE Intl. Performance, Computing and Communications Conf.*, Phoenix, Arizona, February 2000.

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

[86] Y. Yemini and S. daSilva. Towards programmable networks. In *The 7th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, L'Aquila, Italy, October 1996.