Distributing Media Transformation Over Multiple Media Gateways

Wei Tsang Ooi
Department of Computer Science
Cornell University
weitsang@cs.cornell.edu

Robbert van Renesse
Department of Computer Science
Cornell University
rvr@cs.cornell.edu

ABSTRACT

Media gateways have been proposed as a solution to the network heterogeneity problem in media multicasting. Services on the gateways transform media streams as they flow through the gateways. In this paper, we present our work on composable services in media gateways. A user can request a computation to be performed on a set of media streams. The system then distributes the computation over multiple gateways for execution. We present an algorithm for decomposing the computation into sub-computations, and an application-level protocol that locates appropriate media gateways to run these sub-computations.

1. INTRODUCTION

Network and host heterogeneity causes problems for delivering multicast media streaming over the Internet. Heterogeneity often forces media streams to be simulcast in multiple formats and varying bandwidth (as many video webcasts are doing), or being multicast using the lowest bandwidth acceptable by all users, sacrificing quality. End-to-end solutions such as Receiver-driven Layered Multicast [12] have been proposed to address the bandwidth heterogeneity problem, but they do not solve format or host capacity incompatibility issues. An alternative solution is media processing in the network. This solution uses entities inside the network to transform media streams – for example, into lower bit-rate streams for slow links, or into another format the receiver is capable of decoding. These entities can be deployed, for example, by ISPs across the Internet, or by corporations within their VPNs.

Various research work has been done in this area. The MeGa [1] media gateway can transcode RTP video streams in multi-megabit MJPEG format on the Bay Area Gigabit Network into 128 Kbps H261 video streams suitable for MBone sessions. Yeadon et al. [19] present a set of QoS Filters that can resize video frames, reduce frame rates and mix multiple video streams inside network switches.

Our work in the Degas system [14] extends the existing media gateway architecture in two ways. First, the Degas gateway is extensible. Degas allows a user to specify a computation to be performed on media streams by submitting a small script into a gateway. Second, Degas tries to minimize bandwidth consumption by assigning computations to appropriate gateways. For example, a computation that transforms media stream to lower bit-rate is assigned to a gateway near the sender, while a computation that increases bit-rate is assigned to a gateway near the receiver.

1.1 Composable Service

Amir et al. introduce the notion of *composable services* for media gateways in [2]. 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 1(a)). These two operations form a linear pipeline. More complicate 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 merges 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 1(b)).

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 better load balancing, and higher throughput when a single gateway becomes a bottleneck. 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-computation to be shared by different users requesting different services. For example, if the transcoding service in Figure 1(a) shared a common source with the "quad-

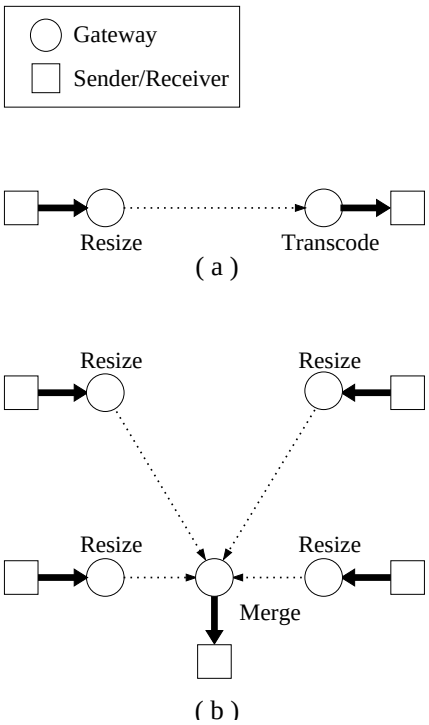

Figure 1: Examples of composable services.

splitter" service, then the output from the scaling gateway could be shared by both services.

In this paper, we present our preliminary work on composable services in media gateways. The work we present in this paper is an extension of our previous work on Degas.

1.2 Degas

Degas is an application-level media gateway system that can perform transcoding, filtering and mixing on video and audio streams of an RTP multicast session. 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. A user who is interested in receiving videos from a session through Degas runs a Degas client. The client first requests a service from Degas. A gateway is selected to serve the client. The selected gateway joins the session requested by the client and runs the computation. The processed video stream is sent to a new multicast session, which the client is listening to. The gateway is selected in a manner that minimizes bandwidth consumption through an announce-listen based protocol called Adaptive Gateway Location Protocol (AGLP). The gateway that serves the client can change dynamically depending on the environment. For example, congestion along the path between the serving gateway and the client can cause the service for the client to be migrated to another gateway.

1.3 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 computation 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 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 computation on the media streams into one *main computation* and multiple sub-computations. The sub-computations are submitted to other gateways for execution. A gateway that runs sub-computations is called a *helper gateway*. The main computation 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.

1.4 Assumption and Constraints

We assume that a helper gateway can subscribe to any subset of sources in a multicast session, since a sub-computation 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 [9] and Internet Group Management Protocol (IGMP) version 3 [5], which currently is an IETF draft.

There is a major disadvantage in sending a stream though multiple gateways – the latency between the sources and the receiver increases because of the decoding and encoding operations that need to be performed at each gateway along a pipeline. [14] shows 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. We can constraint the system to split off a sub-computation 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, computation needs to be re-decomposed and re-assigned. An example of frequently changing computations is one that filters input streams depending on who is the current speaker of a teleconferencing session.

1.5 Research Goals

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

A second concern is maximizing sharing between different services. A gateway can take sub-computations that are already running on other gateways into consideration and try to share those services if they share the same sources and operations. Another concern is network bandwidth consumption. We should distribute the computation 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-computation 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 paper, we focus only on 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 computations, 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: computation 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-computations, and (2), assign sub-computations to gateways with the goal of reducing bandwidth consumption.

1.6 Paper Outline

The rest of the paper is organized as follows. Section 2 describes our algorithm for splitting computation into sub-computations. Section 3 describes how we use a decentralized protocol to locate gateways and assign sub-computations to them. We present performance results of our protocol in Section 4 and 5. We discuss related work in Section 6 and conclude in Section 7.

2. COMPUTATION DECOMPOSITION

In this section, we describe the algorithm we use to split a computation into sub-computations. As we decouple the problem of decomposing computation 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-computation. Our algorithm uses the estimated size of compressed videos as a parameter to decide how to split a computation, 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-computation that is assigned to it using the same algorithm. These sub-sub-computations can then be spawned off by the helper gateway to other gateways for execution.

2.1 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. This is a natural representation of a computation, and has been used in many video processing softwares (e.g. Rivl [17], PSVP [11]).

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 $w : E_G \rightarrow R^+$. 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 $u \in T$ and $v \in S$. The set of all edges that cross a cut (S, T) is called a *cut-set* for (S, T). A computation is split into sub-computations 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-computation. The sub-computation that contains V_{root} will be the main computation. See Figure 2 for an example.

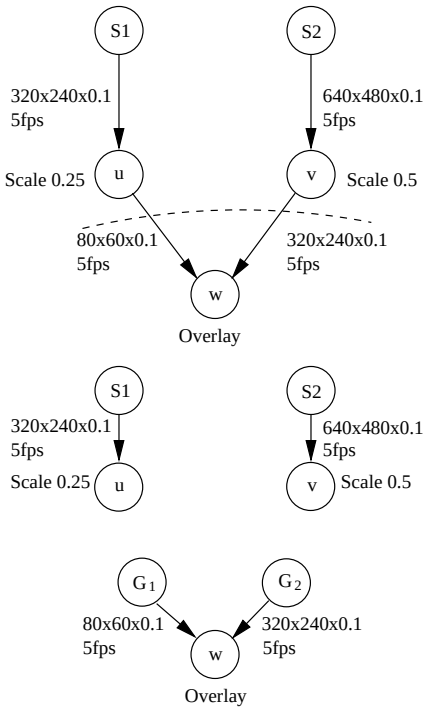

Figure 2: The top diagram shows an example of a computation tree. Edges (u, w) and (v, w) divides the computation tree into three smaller computation trees, which corresponds to the sub-computations.

As we assume that the sub-computations will be assigned to different gateways for execution, the weight of the edges

across the cut corresponds to the amount of 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 3 shows our algorithm for finding the minimum cut-set (or *mincut*).

```
MinCut(G)
1.  E_cut ← {}
2.  for each subtree G_i of root(G) do
3.      if G_i is a single node then
4.          E_cut ← E_cut ∪ {(root(G_i), root(G))}
5.      else
6.          E'_cut ← MinCut(G_i)
7.          if cost(E'_cut) > w(root(G_i), root(G))
8.              E_cut ← E_cut ∪ {(root(G_i), root(G))}
9.          else
10.             E_cut ← E_cut ∪ E'_cut
11.return E_cut
```

Figure 3: Algorithm for finding mincut 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 will skip the formal proof of the correctness of the algorithm, and only provide an outline of the 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 mincut 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 mincut of the 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 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, lines 6 - 10 correctly find the edges that belongs to the mincut of G. By induction, we conclude that MinCut correctly finds the mincut of computation tree G.

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

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

3. GATEWAY LOCATION PROTOCOL

In this section, we describe how a main gateway locates and assigns sub-computations to helper gateways. Our protocol extends our previous work on Adaptive Gateway Location Protocol (AGLP), and therefore we first describe how AGLP works. A brief description is given in the next section. Details about the protocol can be found in [13].

3.1 How AGLP works

AGLP is used in Degas to locate a single gateway to service a client, with the goal of minimizing bandwidth consumption. It is an application-level, soft state ("hints") protocol. The simplicity of the model allows us to build a scalable, robust protocol that is resilient to crashes and message loss. By using soft state, AGLP can adapt to changing network conditions, as well as the birth and death of gateways, senders, and receivers. AGLP can migrate computations between gateways to adapt when the environment changes in order to reduce network congestion.

AGLP uses propagation time as a parameter to decide if a gateway is suitable to service a client. It does not take geographical locations, topology or number of hops into consideration. Previous study [4] shows that there is little correlation between these parameters. We use propagation time as this directly corresponds to end-to-end delay, the parameter we care most about.

In the design of AGLP, we assume that sources, gateways and clients run NTP to synchronize their clocks to measure propagation time between hosts. However, in the absence of NTP, we can use other schemes to estimate propagation delay, such as SPAND [16], or simply ping.

In AGLP, all clients and gateways communicate on a common, well-known multicast channel. AGLP consists of two phases. The first phase, *Quick-Start Phase*, chooses a gateway that is close to the client to reduce start-up latency, without worrying about optimizing bandwidth consumption. The second phase, the *Adapting Phase*, optimizes the bandwidth utilization by migrating services to better gateways.

At the beginning of the Quick-Start Phase, a client C who wants to request some computation to be done on some input streams multicast a request message onto the common multicast channel. A gateway G_i that receives the request message and is available to serve C replies with a offer message. Instead of replying immediately, each G_i waits for some time before multicasting the offer. A gateway will suppress its offer message if it has received an offer message from another gateway while waiting. Client C listens and accepts the first offer that it receives. Subsequent offers from other gateways will be ignored by C.

After the client chooses the gateway, the client and the chosen gateway, denoted G_0, periodically multicast a serve and served-by messages to indicate that G_0 is currently serving the client.

After joining session s, G_0 starts to collect information about the session. This information includes the identity of the senders $S_0..S_k$, bandwidth of the input streams $b_0..b_k$ and the output stream b_C, and the distance (or latency) $d_{0,j}$

from each sender S_j. This information is included in the serve messages and is multicast to every other gateway.

Each gateway G_i, that is available to serve C, maintains a soft state table of distances to itself from the sources, $d_{i,0}..d_{i,k}$ and the client $d_{i,C}$. The table is refreshed by periodically joining the RTCP [15] session of s, listening to RTCP packets, and calculating the distance by subtracting the NTP timestamp of a sender's report from the arrival time.

Each gateway periodically evaluates its suitability of serving client C by calculating a *score*, x_i 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 of a gateway G_i. 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 since it will consume less bandwidth. Each gateway with a score larger than a threshold ϵ will try to replace the current gateway. Gateway G_i waits for $T_{\text{replace},i}$ seconds, and multicasts a replace message, containing its score x_i. If G_i receives another replace message from another gateway with higher score, G_i suppresses its own replace message. $T_{\text{replace},i}$ is set inversely proportional to x_i, so that a gateway with a high score replies quickly. The current gateway keep tracks of the gateway with the lowest score so far, denoted G_r. T_{adapt} seconds after G_0 receives the first replace message, gateway G_0 multicasts a handoff message and sends the computation to G_r. G_r subsequently starts the service, sends the output stream to a new group, and multicasts handoff-ok.

G_r begins to multicast serve messages periodically. Upon receiving both handoff-ok and serve from G_r, C knows that another more suitable gateway has been found. C switches to the output group of G_r. C stops announcing served-by for G_0 and starts announcing served-by for G_r. As G_0 no longer receives served-by from C, it eventually stops processing video streams for C after a timeout.

3.2 Extension to 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 then adapt), we can do it in one since startup latency is no longer a concern.

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

It is important that we defer the Adapting Phase until all the helper gateways are identified and initialized. The optimal locations to execute main computation and sub-computations

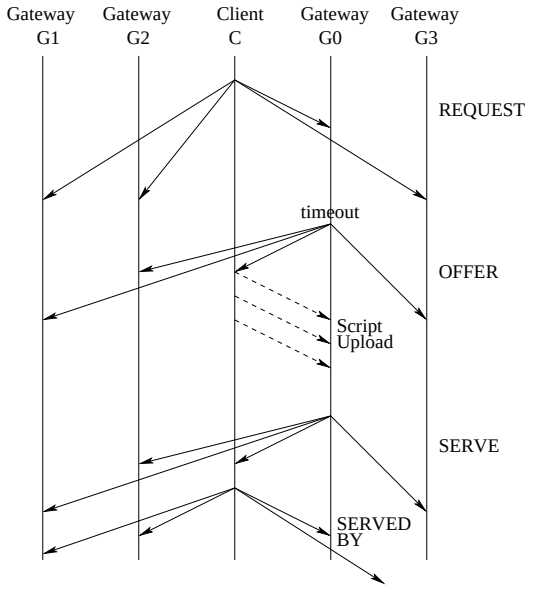

Figure 4: The Quick-Start Phase of AGLP.

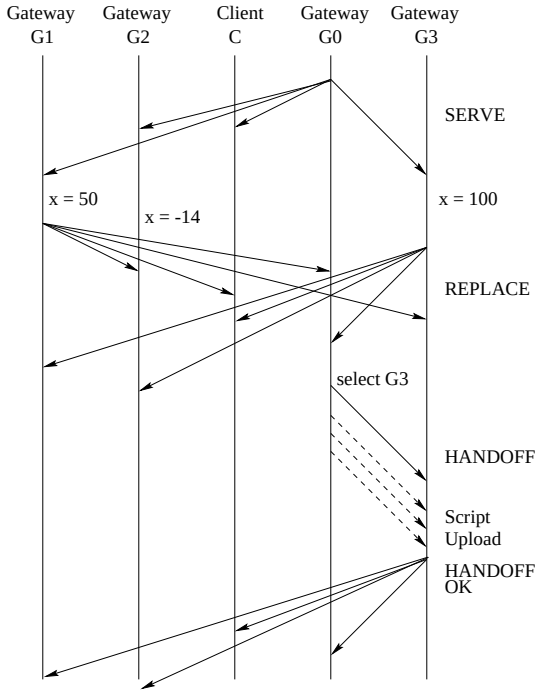

Figure 5: The Adapting Phase of AGLP.

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-computation to be executed on a gateway, including the input session and identities of the sources to the sub-computation, 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-computation 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-computation. 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-computation, and define b'_0 be the output bandwidth of the sub-computation. 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-computation. Let G_h be the one selected. G_0 then multicasts a help-accepted message, and hands off the sub-computation 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-computations, G_0 continues to process the input streams using the un-decomposed computation. Once G_0 knows that G_h is ready, G_0 reconfigures its computation by removing the subtree that corresponds to the sub-computation 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-computation to a better helper, and G_0 will switch its input from G_h to the new helper.

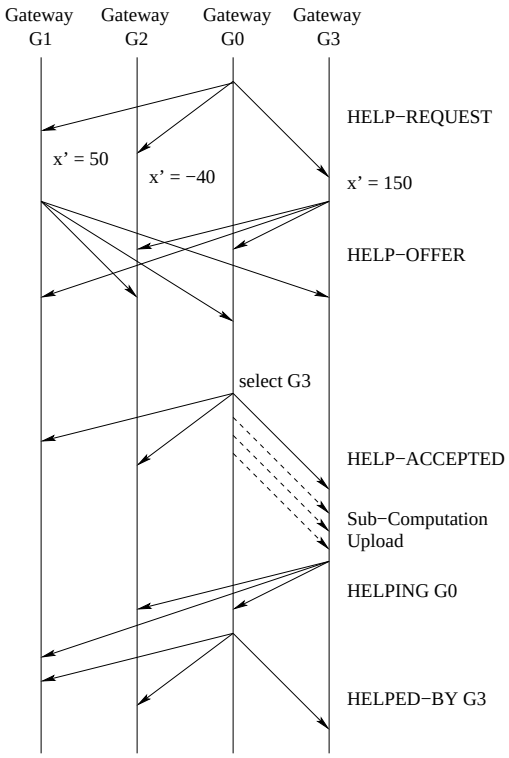

Figure 6: The Splitting Phase of AGLP.

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

4. 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 [3] and ran simulations on randomly generated 100-node networks using the gt-itm random topology generator [6]. We used a computation similar to Figure 1(b) with three sources. The computation is decomposed into three sub-computations that resize the input streams, and a main computation that merges the stream. In this section, we present our simulation results.

Figure 7 shows the number of migrations of the main gateway for the original AGLP (labeled AGLP), and the extended AGLP (labeled AGLP++). The result shows that the average number of migrations for extended AGLP is slightly less in the original version. This can be explained as follows. The computation we used for this simulation is

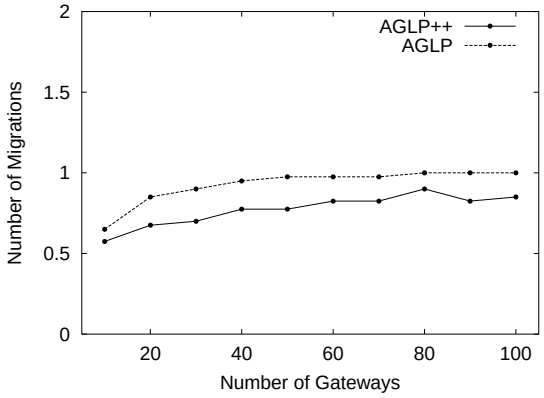

Figure 7: Number of Migrations vs Number of Gateways.

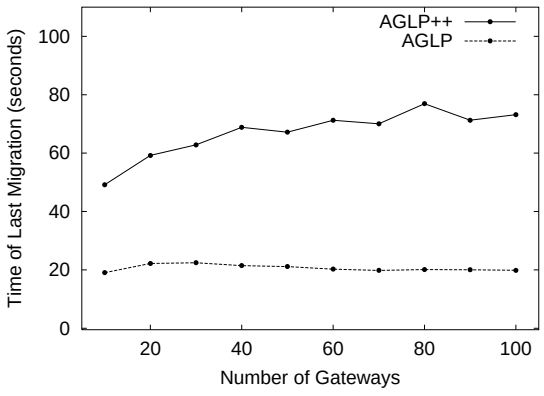

Figure 8: Time of Last Migration vs Number of Gateways.

one that reduces bandwidth consumption, and therefore is best ran close to the sources. Thus it is more likely that we need to migrate the computation from the gateway we pick in the Quick-Start Phase to another gateway close to the sources. In the case where we divide the computation into sub-computations, the main computation merges streams, producing an output stream that uses more bandwidth than the total input bandwidths. Hence the main computation is best ran near the receiver. There is a good chance that the gateway we pick in Quick-Start Phase, which is the gateway closest to the receiver, is already good enough and no further migration is needed in this case. Hence fewer migrations are needed in extended AGLP.

However, as we deferred the Adapting Phase until we assigned sub-computations to helper gateways, the time it takes to migrate the main computation to the optimal gateway increases significantly. Figure 8 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 30 - 50 seconds.

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 9 and Figure 10.

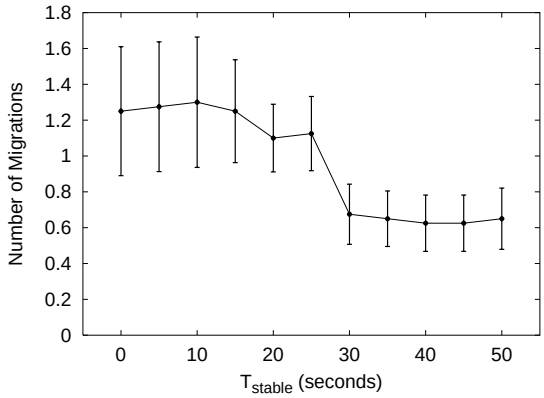

Figure 9: Average Number of Migrations vs T_{stable} for 20 gateways, with 95% confidence interval.

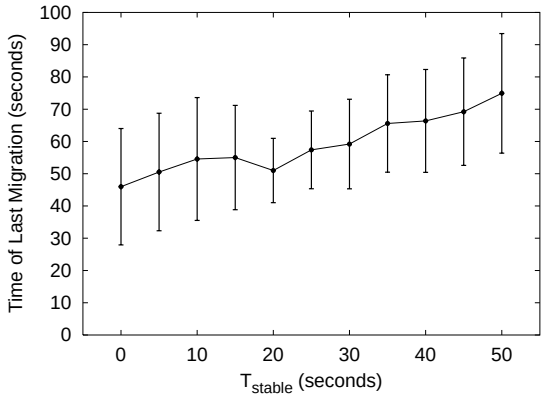

Figure 10: Average Time of Last Migration vs T_{stable} for 20 gateways, with 95% confidence interval.

An interesting observation from Figure 9 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-computations are assigned to helper gateways, and may unnecessarily migrate the main computation to a gateway near the sources.

Figure 11 shows the average number of times helper gateways start executing sub-computations. As the main gateway in our simulation requested help for three sub-computations, a value of 3 indicates that all sub-computations are assigned to their respective optimal helper gateway the first time. A value larger than 3 implies some migrations of sub-computations. This graph shows that the number of migrations per sub-computation is less than 1, indicating that we are able to locate good helper gateways to run the sub-computations most of the time.

Figure 12 shows the average number of help-offer messages that the main gateway receives per number of sub-computations. Just as the replace messages during the Adapting Phase, we use multicast damping to avoid feedback implosion. The graph shows that this number increases very slowly as the number of gateways goes from 10 to 100. A

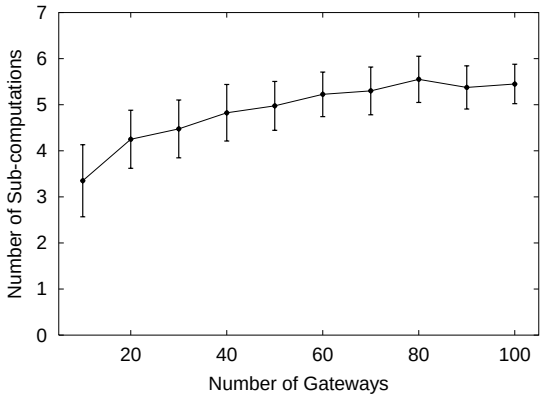

Figure 11: Average Number of Sub-Computation Executions vs Number of Gateways, with 95% confidence interval.

caveat here is that this graph shows the number of messages per sub-computation. The number of help-offer increases linearly with number of sub-computations, and hence does not scale well. One solution for this problem is to request help for the sub-computations 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.

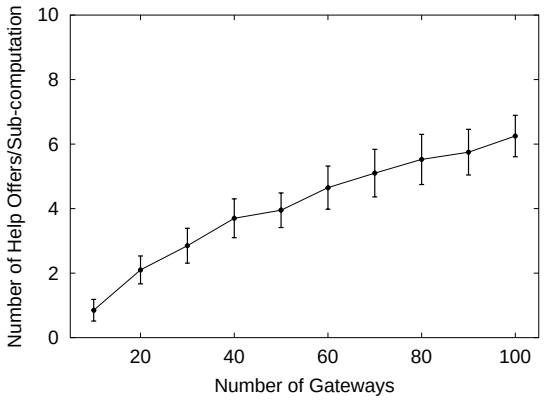

Figure 12: Average Number of Help Messages vs Number of Gateways, with 95% confidence interval.

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.

5. 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 experiments are carried out using a prototype of the Degas system [14] on a local area network. The computation that we used in these experiments is shown in Figure 13. This computation merges two high quality MJPEG streams into a low-quality, 5 fps H.261 stream.

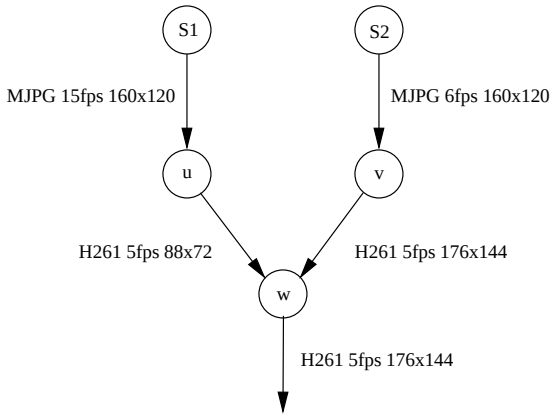

Figure 13: A picture-in-picture computation used in experiments.

We first ran the computation 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 distributed the same computation onto three gateways. We used two Sun Ultra-80 computing servers as helper gateways to run operation u and v, and run operation w on host A as the main gateway.

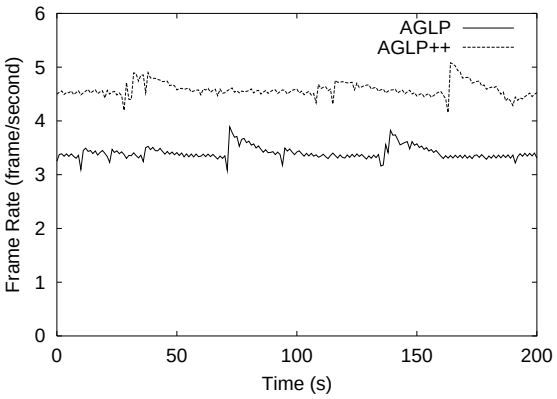

Figure 14: Frame rate versus Time.

Figure 14 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-computations 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 15 shows the corresponding data rate received in the experiments.

We measured the period between rendering of frames for both experiments. The results are shown in Figure 16 and Figure 17. Our result shows that we are able to reduce jitters 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 computation on multiple gateways to be

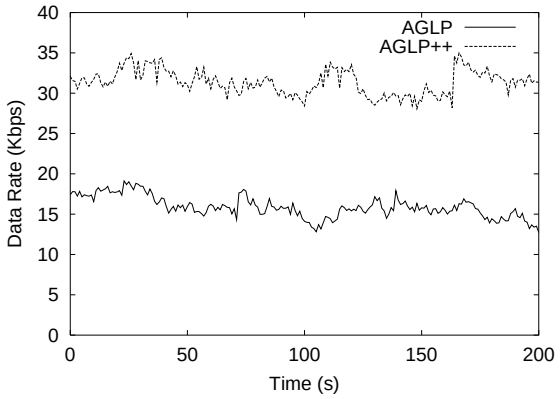

Figure 15: Data rate versus Time.

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 computation 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.

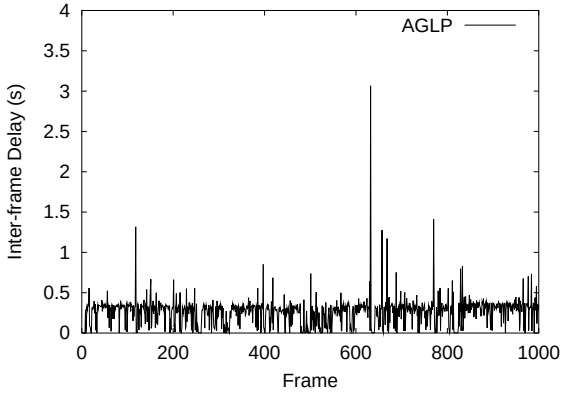

Figure 16: Inter-frame Rendering Delay versus Frame Number for Single Gateway.

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 jitters and end-to-end delays, thus improving the quality of the video streams received by the user.

6. RELATED WORK

Media processing in the network has been proposed and studied in [1, 18, 19]. These previous proposals concentrate on running media processing in a single location in the network.

Distributing video processing across multiple nodes in the network has been studied in [11, 10]. 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 send different frames to different hosts

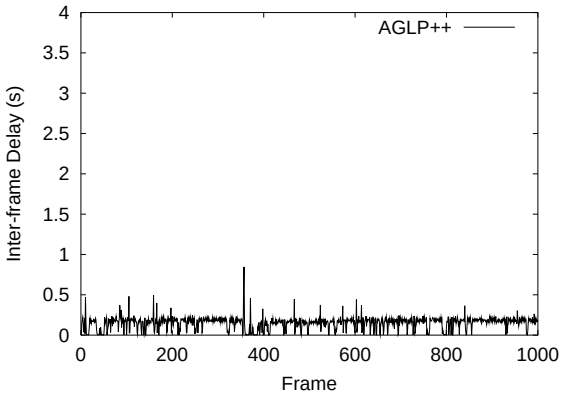

Figure 17: Inter-frame Rendering Delay versus Frame Number for Multiple Gateways.

for processing. The results are then sent to another host, which multiplexes it 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. In contrast, our work focuses on functional parallelism, where multiple hosts perform different operations on the video streams as it flows through the hosts.

The concept of functional decomposition was proposed in [2] in the context of the Active Service framework. A media gateway runs as a service, but 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. Our work is different from this in that our service composition is done automatically and is transparent to the client, and we take network bandwidth into consideration when we decide where to run the services.

Several recent wide-area network services allow composition of their services as well. CANS [7] (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 [8] 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. There are two main differences between these two approaches and ours. First, they attempt to construct a path using *available* mobile code to meet a user's request. In contrast, our mechanism decomposes a user's request into blocks of mobile code. Second, both Ninja and CANS use centralized algorithms with global knowledge (link bandwidths and load on servers) to construct the path. AGLP++ uses a decentralized approach that does not assume such knowledge.

7. CONCLUSION

We presented our initial work on composable services in media gateways. A computation on a media stream is divided

into multiple sub-computations and is sent to multiple gateways for execution. In this paper, we focus on two fundamental design issues, how a computation can be decomposed, and how to assign the sub-computations to gateways. There are still many open issues that remain to be studied.

We plan to improve our system to allow a computation that changes frequently to be distributed onto multiple gateways. This is a common class of computations, because multicast sessions are dynamic in nature. It is also important because currently our gateways manage resources by modifying the computation when resources are low, for example, by reducing frame rate, resolutions or changing the output to gray scale. The research issues involved include how to communicate the changes to helper gateways, and how to re-assign the helper gateways efficiently when changes occur.

We also plan to look at the global picture of services running on all gateways. An interesting issue is how sub-computations with the same operation and inputs can be identified and shared among multiple services. Another issue we plan to look at is fairness among the services. We want to assign the sub-computations such that each user gets a fair share of overall resources in the gateways. The challenge is to do this in a decentralized and scalable manner.

8. ACKNOWLEDGMENTS

This research was supported by DARPA/AFRL (contract F30602-99-1-0532). We like to thank Ketan Mayer-Patel and the anonymous reviewers for their constructive comments and useful suggestions on this paper.

9. REFERENCES

[1] 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.

[2] 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.

[3] S. Bajaj and et al. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999.

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

[5] 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.

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

[7] 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.

[8] S. D. Gribble and et al. The Ninja architecture for robust internet-scale systems and services. *Computer Networks (Special Issue on Pervasive Computing)*, 35(4):473–497, March 2001.

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

[10] 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.

[11] 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.

[12] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM*, pages 117–130, Stanford, California, August 1996.

[13] W. T. Ooi and R. van Renesse. An adaptive protocol for locating media gateways. In *Proceedings of the 8th ACM International Conference on Multimedia*, pages 137–145, Marina del Ray, California, November 2000.

[14] W. T. Ooi, R. van Renesse, and B. C. Smith. The design and implementation of programmable media gateways. In *Proceedings of 10th. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'00)*, Chapel Hill, North Carolina, June 2000.

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

[16] 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.

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

[18] 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.

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