An Active Service Framework and its Application to Real-time Multimedia Transcoding

Elan Amir, Steven McCanne, and Randy Katz
University of California, Berkeley
{elan,mccanne,randy}@EECS.Berkeley.EDU

Abstract

Several recent proposals for an "active networks" architecture advocate the placement of user-defined computation within the network as a key mechanism to enable a wide range of new applications and protocols, including reliable multicast transports, mechanisms to foil denial of service attacks, intra-network real-time signal transcoding, and so forth. This laudable goal, however, creates a number of very difficult research problems, and although a number of pioneering research efforts in active networks have solved some of the preliminary small-scale problems, a large number of wide open problems remain. In this paper, we propose an alternative to active networks that addresses a restricted and more tractable subset of the active-networks design space. Our approach, which we (and others) call "active services", advocates the placement of user-defined computation within the network as with active networks, but unlike active networks preserves all of the routing and forwarding semantics of current Internet architecture by restricting the computation environment to the application layer. Because active services do not require changes to the Internet architecture, they can be deployed incrementally in today's Internet.

We believe that many of the applications and protocols targeted by the active networks initiative can be solved with active services and, toward this end, we propose herein a specific architecture for an active service and develop one such service in detail — the Media Gateway (MeGa) service — that exploits this architecture. In defining our active service, we encountered six key problems — service location, service control, service management, service attachment, service composition, and the definition of the service environment — and have crafted solutions for these problems in the context of the MeGa service. To verify our design, we implemented and fielded MeGa on the UC Berkeley campus, where it has been used regularly for several months by real users who connect via ISDN to an "on-line classroom". Our initial experience indicates that our active services prototype provides a very flexible and programmable platform for intra-network computation that strikes a good balance between the flexibility of the active networks architecture and the practical constraints of incremental deployment in the current Internet.

1 Introduction

One of the key strengths of the Internet service model is that it abstracts away the details of how messages are forwarded through the network. On the one hand, this design principle is extremely powerful because it divorces applications from much of the complexity of the underlying communication system, but at the same time, it constrains applications because they cannot exploit detailed knowledge of the underlying network to enhance their performance. One application class that is constrained in this way are so-called "video gateways" [3], which are computational elements that adjust the bit rate of video stream or collection of video streams to accommodate the constrained capacity of communication links at strategic locations within the network. Because video gateways perform their optimization directly on the underlying data stream from within the network itself, they must be physically situated at an appropriate, perhaps arbitrary, point within the network. However, even though these agents are deeply embedded within the network infrastructure, they are actually created, configured, and controlled dynamically by the user application on the end-system at the edge of the network, for instance, by employing application-specific protocols to convey "receiver interest" into the network to best configure the agent for the receiving user's preferences and capabilities [2].

Unfortunately, the Internet service model has no native support for deploying agents within the network in this fashion. To overcome this limitation, the "Active Networks" initiative [30] proposes that the Internet service model be replaced with a new architecture in which the network as a whole becomes a fully programmable computational environment. In this framework, not only do application-defined entities run on arbitrary nodes in the network but individual packets can be programmed to perform arbitrary actions as they propagate through the network — "programmability" migrates down the protocol stack from the application layer across the transport layer and into the network and datalink layers.

While the requirements of video gateways are often cited in the active networks literature as proof of a pressing need for this new infrastructure, the far-reaching implications of uprooting and supplanting over twenty years of Internet design experience begs the question: Is active networks both *sufficient and necessary* for deploying scalable, flexible, and robust services, like the video gateway service, within the network on behalf of the user? While we concede that a comprehensive active networks framework would immediately solve this problem and is therefore *sufficient*, we argue that for a large and important sub-class of the active networks design space — namely those applications like the video gateway that provide an application-level service — the active networks architecture is not strictly *necessary*. Instead, we argue that these applications are adequately and effectively supported by a "programmable service architecture," built on top of and in harmony with the existing Internet service model, that allows users to download and run code on their behalf at strategic locations within the network. We and others call this programmable network infrastructure *active services* because we restrict the design to application-level *services* yet we inherit the novelty of the *active* networks programmability[16].

While we believe that the active services framework holds the promise to enable many active networks-like applications, we do not propose this framework as an outright replacement for active networks because it cannot possibly meet all of the goals and support all of the applications targeted by active networks. The ability to quickly re-program and deploy new protocols at any level in the network architecture is not possible in nor is it the goal of active services, and applications like nomadic routing [22], localized TCP optimizations (e.g., "snoop" [8]), SYN-flooding avoidance, etc. are not naturally amenable to our framework. Nevertheless, we believe that the active service framework is an important new networking technology not only because solving active networks sub-problems in this domain will contribute to the overall understanding of the active networks problem, but also because it is a valuable and useful technology in its own right — active services can solve a number of pressing networking problems in a way that is fully compatible with today's Internet. In fact, we have built, deployed, and extensively exercised an experimental active service framework — the media gateway system described later in this paper — that has provided a tangible application to a real user community who used it to do real work and collaboration on a regular basis for the past six months.

In the remainder of this paper, we develop the design rationale for our active research framework and present a detailed overview of its architecture and use. The next section describes our active service framework. Section 3 describes the design of a prototype service that runs on our framework: the MeGa service for media gateway deployment. Section 4 presents implementation details. In Section 5 we present a brief survey of related work and then conclude.

2 An Active Service Framework

A video gateway mitigates bandwidth heterogeneity by adjusting each video flow's bit-rate in a controlled fashion to meet each link's available capacity [3, 32]. While mechanisms to accomplish such rate adaptation are well understood, companion mechanisms for instantiating the transcoding agents at the appropriate places in the network and for flexibly configuring and controlling them once running are comparatively underdeveloped. After a number of years of experimentation with techniques for deploying and configuring video gateways and drawing upon emerging ideas in active networks and the success of Java's platform-independent computing model [2, 3], we refined our creation and control protocols into a programmable infrastructure that became the active services prototype presented herein. Because we believe our service architecture is useful for a broad range of applications beyond simple video transcoding, we developed a re-usable framework where the "active" multimedia gateway service consists of two key levels of abstraction:

- an active service framework, which provides the programmable substrate on which to build arbitrary network services, and
- the specialization of that framework for the particular problem at hand, in our case, a media gateway service.

In this section, we describe a specific design for an action service that we call "AS1" (i.e., "active service version 1") and in the subsequent section we describe the specialization of AS1 for the particular task of media transcoding manifested in our Media Gateway (MeGa) active service.

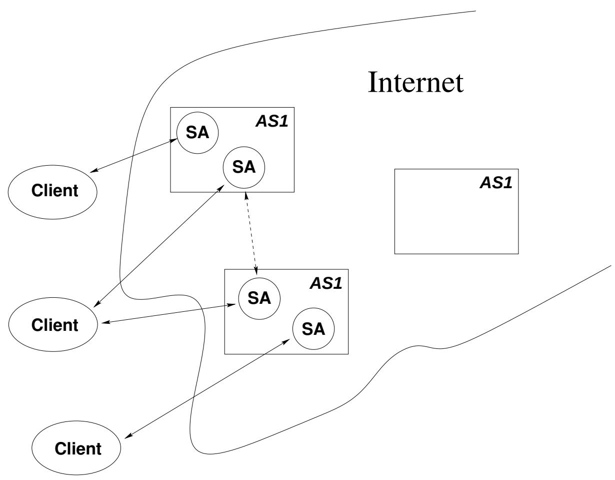

Figure 1: The AS1 active service framework: *clients* rendezvous with AS1 *clusters* via well-known point-of-contact unicast or multicast addresses to create computational entities called "service agents" or *servents* that implement the desired service. A client can simultaneously interact with multiple clusters to create and interpose servents across the network to effect more complex services.

The key abstractions of and vocabulary for our AS1 framework are illustrated in Figure 1. Users that require an active service are called *clients* and within the network reside one or more pools of active service nodes each called a *cluster*. Clients instantiate application-level service agents, called *servents* (i.e., the contraction of serv<u>ice</u> ag<u>ent</u>) on one or more clusters. A servent's behavior is defined by its arbitrary active-service program, and once created, the servent is controlled by the clients in a servent-specific fashion, i.e., the control implementation is embedded in the servent definition. Multiple servents may be composed either within the same cluster or across different cluster to implement more complex services. To support robust operation and graceful reaction to system failures, the servent's ongoing existence is continually refreshed by the client, and when refresh messages cease, AS1 detects the lost client and terminates the corresponding servent.

To facilitate the design process of the AS1 architecture, we decomposed the framework into six core but interdependent components:

- **Service Environment**: The service environment defines the "active" part of active services, i.e., the programming model and interfaces to programming resources available to the servents that run within AS1.
- **Service Location**: To instantiate a new servent, the client must first locate an AS1 cluster. The service location subsystem provides a mechanism for the client to rendezvous with AS1.
- **Service Management**: The service management sub-system allocates the finite computational resources of the cluster across servents to implement load balancing, mechanisms for graceful degradation under high load, and admission control.
- **Service Control**: Once a servent is instantiated on AS1, the client must be able to dynamically re-configure and control it via the service control subsystem.

- **Service Attachment** : If a client is does not have direct access to the service infrastructure (i.e., because of lack of multicast support or because of an administrative firewall), a client utilizes the service attachment sub-system to "tunnel" through the barrier for general network connectivity.
- **Service Composition**: Some services are best implemented as a collection of agents spread across the network. The service composition sub-system allows clients to contact multiple service clusters and interconnect servents running within and across clusters.

To illustrate the role of each component within our active service framework, we will outline a simple scenario where a bandwidth-impoverished, multicast-incapable client wishes to join a global multicast video session. To initiate the process, the client contacts the active service through a well-defined rendezvous mechanism (service location), which by default employs an exchange of messages over a well-known multicast address. But since the client is not directly attached to a multicast-capable network, it must rely upon a unicast point-of-contact, which it might obtain from DNS, DHCP, or some other service location protocol. This point-of-contact information returned by this query includes a list of addresses so that the client can "round robin" over multiple contact points for simple load balancing and fault recovery. The client then submits its service request on the rendezvous address in the form of a program (service environment) that defines the servent to be instantiated, namely a video gateway. In response, AS1 creates a single instance of the specified video gateway according to its internal load-balancing algorithms (service management). For the duration of the session, the client runs a servent-defined protocol to dynamically control transcoding parameters of the gateway; that is, the control protocol is embedded in the active program running on the service and its semantics are orthogonal to and independent of AS1. Finally, the client may create multiple gateways on multiple multicast groups chaining them together to accommodate a range of bandwidth requirements (service composition). While our architecture admits such configurations, we have not yet worked out all the details of service composition, how clients and/or servents would interact with the session directory service [18] to allocate multicast address, and so forth; these problems are topics of future work.

In the remainder of this section, we detail the design of five of the six components in the AS1 active service framework, with the exception of service composition, which remains a topic of future work.

2.1 Service Environment

A crucial capability of active services is flexible and simple deployment of application-level computation within the network. To this end, we follow the lead of other projects in this area [16, 17, 34] by implementing an environment that consists of a programmable substrate that the servents program to implement complex computation. But we diverge from the active networks approach by constraining our environment to the application layer. That is, the environment does not allow the servent to manipulate routing tables, forwarding functions, network management functions, etc. We believe that this constrained approach strikes a good balance between the flexibility of the active networks architecture and the practical constraints of incremental deployment in the current Internet.

Because our principal research efforts revolve around real-time multimedia networking applications, we implemented our AS1 programmable substrate using the "MASH platform" [23]. MASH is a Tcl [26] interpreter extended with real-time multimedia and networking related capabilities. Tcl provides a simple, flexible and easy-to-use programming model based on scripting, while the interface to the AS1 resources consists of a method API to a set of Object Tcl classes [33] that can be invoked from the servent's Tcl program. Under this model, servents are simply Object Tcl scripts interpreted by MASH, which we call "mashlets". The use of a scripting language for plumbing together components in our programmable substrate strikes a good balance between the power of low-level "system" languages such as C++, and the flexibility and ease of use of high-level "scripted" languages such as Tcl/OTcl [27].

Our experiences with MASH have led us to conclude that a fully general active service programming environment would unduly burden the active service developer and lead to unavoidable performance constraints. In contrast, a domain-specific programming model simplifies the problem of defining a set of APIs to every possible resource that might be required of a servent. Consequently, just as we have narrowed down our design space from active networks to active services, we also believe that each instance of an active service should be further narrowed in scope to a specific domain. In our case, AS1 is tailored for real-time multimedia networking. Other service environments and design decisions might be made to support domains like web caching or application-defined reliable multicast.

Finally, an important property of the service environment not yet addressed in our work is the safety of untrusted, third-party code. While we have not included a solution to this problem in our current design, nothing in our service environment model precludes the use of type-safe languages such as Safe-Tcl [10], Java [5], or Python [28] and as such we view this problem as important, yet orthogonal to our current endeavor. As part of our future work, we plan to leverage ongoing work in the active networks research community to incorporate safe languages into our service environment.

2.2 Service Location

Before a client can instantiate a servent, it must first locate the active service. That is, the client must obtain bootstrap configuration information that enables it to rendezvous with an AS1 cluster.

We have identified two basic approaches to this problem. The first relies on a centralized server at a well-known location that provides the necessary bootstrap. In this approach, the client obtains necessary rendezvous information from a server, e.g., using the Dynamic Host Configuration Protocol (DHCP) [6].

In contrast to this approach where a client "pulls" down service location information from a well-known point, multicast communication can be exploited to "push" service location information out to multicast listeners. In this model, a client listens on a well-known multicast address over which the required configuration information is periodically transmitted. Hodes *et al.* [19] describe a generalized scheme for service location using this approach based on the Service Location Protocol (SLP) [7].

In AS1, we decouple the physical location of the active service from the control communication channel by leveraging the level of indirection offered by multicast communication. This greatly simplifies the service location problem. Now, the AS1 bootstrap requires just a single, *location independent*, piece of information: the network address for this control channel. If clients do not have multicast service, they communicate with the AS1 cluster through application-level forwarders (see Section 2.5) that enable unicast clients to "join" the control multicast group. In this case, we supply the clients with list of addresses of these forwarders or point-of-contacts, e.g., using DHCP. In our prototype, we implemented static configuration for simplicity (e.g., a file in /etc for Unix or a property in the registry for Windows).

2.3 Service Management

Once clients rendezvous with the active service, they can create arbitrary instances of servents within and across the service cluster. But because servents can induce an arbitrary computational load, the cluster must intelligently apportion its computing resources across service requests to properly balance load and gracefully degrade (or deny) service under high load.

We decomposed this service management problem into two subtasks:

- the *processor sharing* task allocates a subset of nodes from a general purpose and shared cluster to act as the active service infrastructure, and
- the *servent creation* task creates servents within the cluster on behalf of a user client.

The remainder of this section develops these two components in detail.

2.3.1 Servent Launching: The Active Service Control Protocol (ASCP)

One approach to resource management in a distributed system such as AS1 is a centralized control model where all control flows through a single "resource manager". Here, each client would issue servent requests to the resource manager that in turn would create servents on processors chosen through some load balancing algorithm. While this approach is relatively simple to construct, it has several disadvantages. First, the existence of a single resource manager presents a single point of failure in the system. Second, it creates a bottleneck in the control path since all resource management decisions must flow through it. Finally, since the state of the system is managed by a single entity — the resource manager — that entity must also implement thorough error detection and recovery, which incurs protocol complexity and implementation overhead.

In contrast to a centralized resource manager approach, AS1 employs a distributed control protocol, the Active Service Control Protocol (ASCP), which does not hold any of the undesirable properties of the centralized approach. Rather, ASCP relies exclusively upon decentralized control through the exploitation of three important protocol building blocks: announce-listen communication, soft state, and multicast damping.

The announce-listen communication model is embodied in a number of common network protocols including the protocol suite that used for multicast-based Internet conferencing applications. These multicast applications assume a communication model where parties in the collaboration session simply "tune-in to" or "tune-out of" the multicast group without any explicit group notification operation. This loosely coupled, light-weight, real-time multimedia communication model is known as the *light-weight sessions* architecture [21].

Announce-listen communication serves as the primary protocol building block for light-weight session applications. The model is characterized by several properties: a shared (multicast) communication channel over which all parties communicate; periodic, self-descriptive (i.e., temporally independent) protocol messages; timer-based aging of state; and reconfigurable components. Together these constructs comprise a communication framework that is particularly robust and resilient with regard to network heterogeneity, scale, and pathologies like communication partitions and packet loss and reordering.

In a sense, announce-listen communication is a form of reliable multicast where sources simply retransmit their data indefinitely. We formalize this view by modeling the state of an announce-listen protocol as a *(key,value)* table. As illustrated in Figure 2, each member of the session maintains its own copy of the table. Each table entry has associated with it an "owner" that identifies the originator of the state contained in the entry. At periodic intervals, each member transmits the entries that it owns. When a member receives an announcement, it inserts the entry in the state table, indexed by the key. If the table does not contain an entry with the same key, or the value of the corresponding entry has changed, the received entry is classified as an *update*. If, on the other hand, there already exists an entry in the table with the same key, the received entry is classified as a *refresh*. In the event that an entry has not been refreshed or updated for a configured period of time, it is removed from the table, a process called *aging*.

The goal of an announce-listen protocol is to maintain consistency across all members' tables. In the presence of plentiful network capacity and low packet loss, this is trivially achieved, but when the announce-listen update rate is constrained or packet loss is non-negligible, inconsistencies between tables will inevitably arise. Nevertheless, inconsistencies are rectified over time by the periodic announcements of the protocol that update the inconsistent entries, thus ensuring eventual consistency and resilience to packet loss. Simultaneously, these updates enable new members joining the announce-listen session (i.e., with an "empty" table) to be quickly brought up to date with the state of the system.

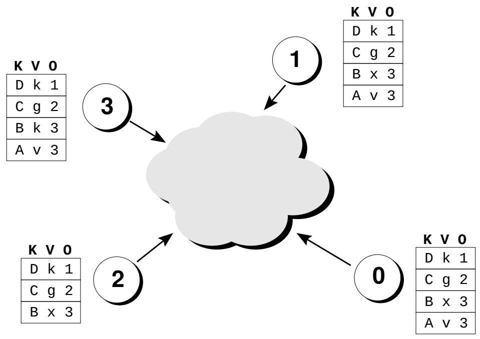

Figure 2: The Announce Listen Protocol Model.

As shown in Figure 2, each member maintains its own copy of the protocol state table and periodically transmits the *(key,value)* pairs (K V) that it owns (O). The figure shows two inconsistencies in the current state of the protocol: member 2 is missing the (A,v) entry from member 3, and the entire group is missing the update to the B entry owned by member 3. These inconsistencies are eventually rectified by the next transmission of member 3's entries.

Announce-listen protocols, in general, combine timer-based aging of state with periodic message refresh and update. This implicitly subsumes both error detection and error recovery. Timer-based aging implies that failures of members in the session cause certain entries of the state table to eventually be deleted while new members or members recovering from failures are quickly brought up to date with the current state by the periodic announcements. Because state retained at each member site in the announce-listen eventually expires but is continuously refreshed it is often called *soft state* [13].

The simplicity and robustness of the announce-listen metaphor make it ideally suited to serve as the bedrock of the ASCP protocol. Figure 3 illustrates how the announce-listen communication model is exploited in ASCP. In this example, the AS1 cluster consists of two hosts and each host runs an agent called a *host manager* (HM). ASCP runs among the HM, the client, and the servent as follows:

(a) In response to a client request, the HM creates an instance of a servent on the local host. As described later, multicast damping ensures that exactly one host manager responds to a client

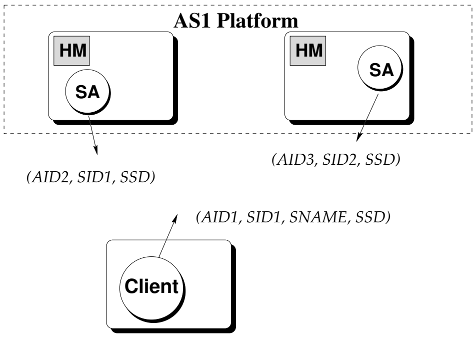

Figure 3: The basic operation of the Active Service Control Protocol, ASCP. Clients announce "requests" for service instances and host managers (HM) respond to these announcements by instantiating a single servent for each unique request. Servents (SA) in turn announce their existence to both the client and to other potential servent sites on the cluster to avoid duplicate servents.

request. Each request carries with it the program (or a reference to the program) that embodies the servent.

(b) The HM notifies the client of the servent's existence.

(c) The client configures the newly created servent as well as itself.

ASCP handles stages (a) and (b) by forming a state table whose *(key,value)* pairs are of the form (S_i, A_j) that designate that the service request S_i is served by the servent A_j. These pairs are sufficient state to inform the HMs as to whether a servent exists for any given client's request. This prevents the HMs from instantiating redundant servents for a given client service request. Consequently, only one servent is instantiated for each unique request.

To construct the state tables, clients and servents each periodically transmit ASCP messages containing the following two pieces of information: an "agent ID" (AID) for the each A_j and a "service instance ID" (SID) for each S_i. The AID is a unique identifier for each servent while the SID is a unique identifier for each instance of a particular service requested by a particular client. SIDs are one-to-one with the servents that are instantiated on the platform and are typically chosen in a way that effects the desired deployment policy across a set of clients. Section 3.1 gives a concrete example of how SIDs are determined with regard to the MeGa service.

A client message contains an additional *Service Name* (SNAME) field, which names the code that implements the requested servent. Since servents are simply Tcl scripts, the SNAME field specifies the name of a MASH script. In our prototype, we recognize two types of service name specifications: location-dependent names, such as URLs, and location-independent names, specifying a generic name of a script, e.g., `vgw-1.0`, for a video gateway. The resolution of a location-independent name to the actual code is performed through an orthogonal mechanism. In our prototype, the HM searches a fixed set of locations, or "script repositories", for the existence of a script with a matching name. Finally, an additional manner in which scripts can be specified is to embed them directly in the ASCP announcement. The disadvantage of this scheme is that it significantly increases the size of the ASCP announcements, and thus the control traffic overhead of the system. Therefore, while in principal this approach is viable, a better approach is to simply specify a location-dependent name that points to the client and then to transmit the script on demand to the requesting HM.

Once a servent is instantiated, it must rendezvous with the client. To this end, ASCP includes a *Service Specific Data* (SSD) field whose role is to convey the initial servent configuration and rendezvous information between the client and servent. After the rendezvous has occurred, client control of the servent can be established and performed independently of ASCP. We demonstrate the use of this field in the context of the MeGa service in Section 3.2.

The soft-state approach to our design yields a particularly robust system with a relatively simple implementation. We illustrate this robustness by describing how the announce-listen/soft-state framework recovers from the three primary sources of system failure in ASCP:

- **Network Failure**: After a failure of network connectivity and a subsequent recovery, the system automatically heals itself since client and servent announcements are self-descriptive and simply update and refresh the ASCP state tables. Any servents that have been terminated due to aged state are promptly restarted just as if they were created from scratch.

- **Servent Failures**: If a servent fails, the state representing that servent will time out, and subsequently the next client announcement serve as a service request to the cluster.

- **Client Failures**: If a client fails, its ASCP messages subside and the client state in the system will expire, thereby triggering the termination of the servent.

The key property of ASCP and the hallmark of announce-listen protocols is that *there is no distinction between regular protocol operation and error detection and recovery.* This yields a particularly robust and fault tolerant system. In contrast to the explicit and often complex error handling code in many protocols, announce-listen protocols, ASCP included, provide *implicit* error detection and recovery, thereby greatly simplifying the protocol design. In summary, we have shown how to exploit the announce-listen communication model and soft state to instantiate servents across an AS1 cluster without explicit connections between clients and servents.

2.3.2 Servent Floods

Although the decentralization of resource management yields a robust design, it induces a new problem. If each host manager in the cluster creates a servent immediately upon receiving a request, many duplicate servents would be simultaneously created and run. Instead, for each service instance request, the HMs should, as a whole, create *exactly and only one* servent.

This duplication effect, which we call a *servent flood*, is analogous to the well-known *multicast implosion problem* where a synchronous protocol event causes a flood of traffic. For example, if the automatic repeat/request protocol primitive is extended to multicast in a naive fashion, acknowledgment messages from the receivers would concentrate back at the source, resulting in an "implosion" effect. More generally, any sort of control actions taken synchronously across a multicast group can result in implosion.

These implosion effects have been combated in a number of network protocols through a technique called *multicast damping*, first introduced in the IGMP [14] protocol and later used in the multicast version of XTP [12] and the SRM reliable multicast protocol [15]. The key to multicast damping is that responders wait a random time interval before acting. After the random wait, the responder multicasts its message to the group. If a responder sees an equivalent message from another member of the group, that responder suppresses its redundant response. In this way, most all the responses from the group are suppressed. The degree of suppression and timeliness of the response are controlled by the probability distribution of response timers.

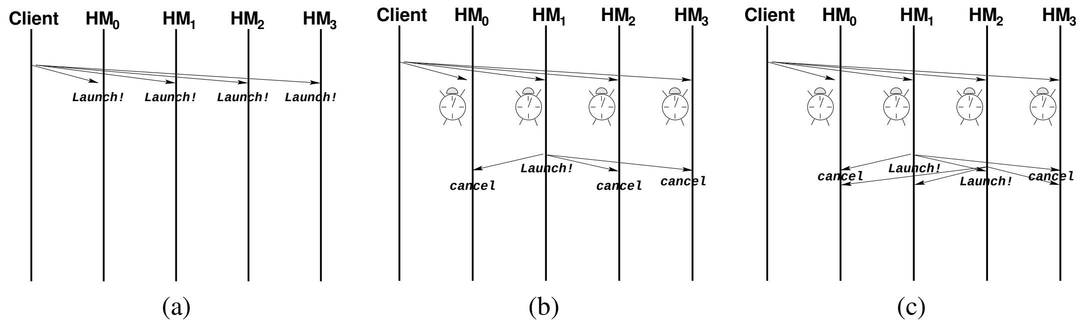

Figure 4: Servent launches. A naive implementation (a) yields launch floods while ASCP prevents servent floods through multicast damping (b). Duplicate gateways can still be launched if the launch timers expire closely enough so that the damping messages are not received within the difference in the timers (c).

In AS1, the analog of a flood of duplicate control messages is a flood of duplicate servents instantiated across the cluster. To avoid this pathology, we employ multicast damping in ASCP as depicted in Figure 4.

Figure 4(b) shows how ASCP avoids servent floods. Upon receipt of a client announcement that requires the creation of a new servent (i.e., the HM does not have any record of a servent handling the service instance requested by the client), each host manager sets a randomized *launch timer*. When the launch timer expires, the HM creates the servent and multicasts a message with the servent's SID. Upon receipt of this message, all other host managers cancel their launch timers, thereby circumventing a servent flood.

The servent flood prevention scheme does not guarantee that all duplicate servents are eliminated. As illustrated in Figure 4(c), when two or more launch timers expire within a round-trip time interval of each other, the damping messages from the HMs do not arrive in time to suppress each other. In this case, redundant servents are created. However, the periodic servent ASCP messages ensure that these servents will learn about each others' existence. The servents can then use a simple, deterministic rule to eliminate all but one of them (e.g., a servent that sees another servent with a larger AID terminates itself).

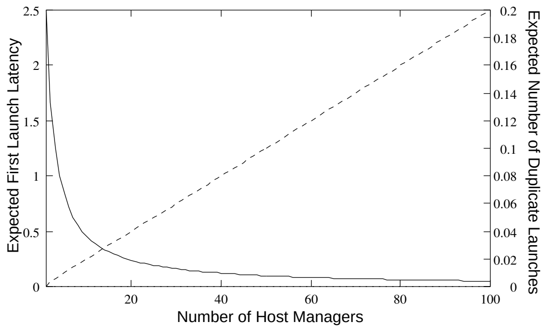

Figure 5: Expected Number of Duplicate Launches (dashed) and Expected First Launch Latency (solid) vs. Number of Host Managers with uniformly distributed launch timers over a five second interval with a transmission latency between HMs of 10 ms.

The problem of minimizing the number of duplicate servents is equivalent to the problem of minimizing the number of duplicate feedback messages in a multicast damping protocol; fortunately, this problem has been extensively studied [15, 29, 25]. In our case, we want to choose the launch timer distribution that minimizes the number of duplicate servents while maintaining acceptable bounds on the mean time until a servent is created. In general, this is a difficult problem, but fortunately for our domain, we can assume that the maximum number of HMs does not exceed a few hundred and that all HMs are on a LAN, i.e., the transmission latencies are constant and small. In this case, a simple uniform probability distribution is adequate. As indicated in Figure 5, the expected number of duplicate launches is low for the range of HMs we are considering and the latency until the servent is created decreases rapidly even for small HM set sizes.

An attractive consequence of the multicast damping scheme is that servents are uniformly distributed across the cluster in a randomized fashion. Thus, even without a centralized resource manager, we achieve a coarse-grained load balancing. Moreover, we can achieve fine-grained load balancing that accounts for measured load on the individual nodes. To this end, each HM monitors the load on its host and biases its launch timer to reflect the measured load. If an HM is lightly loaded, its launch timers are short, but as the load increases, the launch timer distribution is biased toward larger values. In addition, we perform simple admission control by setting the launch timer to infinity when a host is at or near full capacity (as defined by a configured-in target load value). This simple yet powerful mechanism implements load balancing without any additional complexity.

To evaluate our load balancing algorithm, we conducted an experiment where we created a large number of media gateway servents on clusters of varying sizes and recorded the resulting number of duplicates. The result, shown below, indicate an even spread of servents per node and low variance in servent distributions. This confirms our intuition that ASCP can implement reasonable load-balancing policies without the aid of a centralized resource manager.

Nodes	Mean GW/Node	Variance GW/Node
1	9.0	0.00
2	8.5	0.25
3	9.3	0.22
4	8.8	0.56
8	8.5	0.50

2.3.3 Processor Sharing

Structuring active services as a cluster environment is attractive since a cluster is easily expandable and our framework gracefully accommodates such expansion; hence, this approach is scalable. In AS1, we can add a processor to the service by merely running an HM on it. To remove a processor from the service, we simply terminate the HM on that host. Thus, the number of processors allocated to the service is one-to-one with the HM "population" on the cluster.

However, a cluster is often available for general use and we would thus like to share the cluster with other applications and potentially with other active services. A simple way to allocate some fixed number of processors to the service is to statically run that number of HMs on the cluster and monitor the cluster to make sure that these HMs are all running. While straightforward, this approach requires supervision to maintain service availability.

A more desirable alternative is to have the HM population monitor itself and maintain its level of availability on its own. In our model, an HM can perform two population maintenance operations: it can copy itself onto another node and it can terminate itself. Using these primitives, we developed an algorithm to maintain the HM population at a target level using the metaphor of a "birth-death" process.

The core mechanism for our birth-death process is announce-listen communication. In this framework, each HM transmits a periodic announcement, and collectively, the HMs estimate the global population. At randomized *adaptation intervals*, each HM updates its population estimate, n, and compares it with the target, N. This comparison implies an action as follows:

- if $n < N$, the HM forks a copy of itself on a randomly selected host from the set of available hosts on which there is no current HM; the new HM is created with probability p_l,

$$p_l = \min(1, \frac{N}{n} - 1)$$

- if $n > N$, the HM terminates with probability p_t,

$$p_t = 1 - \frac{N}{n}$$

- if $n = N$, the HM does nothing.

Under this framework, if $n < N$ (i.e., there are less HMs than the target), then the number of HMs increases quickly toward the target at the rate prescribed by the adaptation interval until $n > N/2$; at this point, the expected number of new HMs per adaptation interval is $N - n$, thus the population size n quick converges to the target level N. Likewise, if $n > N$, the expected number of HMs that terminate is $n - N$, and we quickly converge to N.

Though this algorithm is decentralized and robust, it can fail with non-zero probability because all HMs could conceivably terminate at precisely the same time. However, this is unlikely because it requires the adaptation intervals of all active HMs to be precisely aligned and further that all these HMs decide simultaneously to terminate. The probability of this event is $(r/T)^n p_t^n$ where T is the adaptation interval and r is the round-trip time between HMs. But,

$$p_t^n = (1 - \frac{N}{n})^n \approx e^{-N} \text{for large } n$$

Thus, we can make the probability of total failure of the system vanishingly small by either adjusting N or by increasing the adaptation interval. Since the HM growth process can run very slowly in the background, we can easily make the adaptation adequately large.

Our algorithm has several desirable properties. First, to populate the cluster, we simply start a single HM on a single host. Next, to add more HMs, we simply increase the target number parameter in the existing host managers. This can easily be performed in a dynamic fashion with network management tools. Similarly, if we want to decrease the number of HMs we decrease the target number and the HM deployment algorithm will remove the excess HMs automatically for us. Removing a machine from the AS1 cluster is trivial since the system will automatically reconfigure itself on the remaining unallocated processors. Similarly, adding a machine to the cluster automatically increases the number of processors available to the system. Finally, since the overall system is based exclusively upon soft-state, the movement and redeployment of HMs on different machines in the cluster does not affect the overall operation of the system. For example, you could decommission a machine with active servents and the system quickly heals itself.

The relationship between the announcement and adaptation interval is critical to the algorithm's performance, since the algorithm depends on the accuracy of the HM population estimate which is computed based on the periodic HM announcements. Increasing the adaptation interval leads to increased "healing" latency in the event of a change of state in the HM deployment, e.g., as a result of a HM failure or change in target number. On the other hand, choosing an adaptation interval that is too small with respect to the announcement interval announcement interval could lead to oscillatory adaptation behavior as a result of instabilities in the control feedback loop.

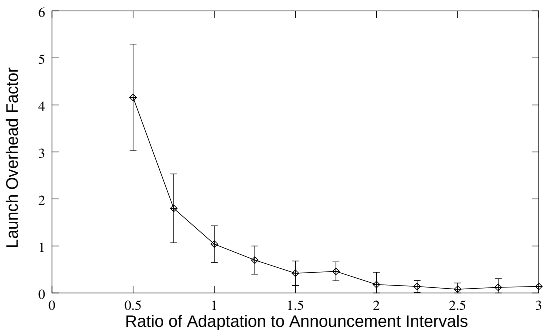

Figure 6: The effects of modifying the ratio of the adaptation to announcement intervals on the launch overhead factor.

To quantify this relationship, we define a metric called the "launch overhead factor" which represents the number of HMs created redundantly before the population converges, divided by N. For example, a launch overhead factor of 1.5 implies that $2.5N$ HMs were created before the algorithm stabilized at N HMs. Figure 6 plots the relationship between the ratio of the adaptation to announcement intervals and the launch overhead factor obtained in an *ns* [24] simulation of the HM deployment algorithm on a cluster of 40 machines with a target population size of 10. The error bars designate the standard deviation of our simulations. As the ratio of adaptation to announcement intervals decreases, the number of redundant launches increases. On the other hand, increasing the ratio decreases the launch overhead, which becomes negligible when the adaptation interval is roughly twice as large as the announcement interval. This makes intuitive sense since this is the smallest adaptation interval large enough to ensure that within it at least one announcement is received from all other HMs. At that point, the performance of the algorithm depends solely on the termination and launch probabilities. Therefore, given an announcement interval, we choose an adaptation interval that is twice as long.

2.4 Service Control

Once a servent has been created, it must be controlled *dynamically* for the duration of the session. By its definition, the service control component of the AS1 architecture is service-specific. That said, in order to maintain the overall robustness of the system, we would like to design the service control protocols according to the same principles of robustness and simplicity. In Section 3.3 we describe an instance of such a protocol in the context of the MeGa service.

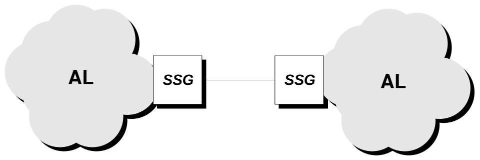

Figure 7: Using soft state gateways (SSG) to bridge two announce-listen control sessions across a bottleneck link.

2.5 Service Attachment: Soft-State Gateways

ASCP relies on multicast communication and a well-known multicast address to rendezvous between clients and the AS1 platform. Clients without multicast service must thus exploit some other mechanism to rendezvous with AS1. Furthermore, a second problem arises if the aggregate bandwidth of the ASCP control traffic might congest a bottleneck link on the path between the AS1 cluster and the client. In this case we must provide mechanism to limit the rate of the ASCP control traffic that is forwarded to the client.

In our design, we solve both these problems through the use of an application-level *soft state gateway* (SSG). The SSG's function is twofold. First, it outputs the incoming ASCP control traffic at a rate that maintains specific bandwidth constraints. Second, it serves as a rendezvous point for clients without multicast service to join the ASCP session. Thus, the SSG serves as alternate mechanism for a client to "attach" to AS1.

Figure 7 illustrates an abstract scenario that demonstrates the functionality of a SSG. Two sessions are connected across a bottleneck link and are both running a common announce-listen (AL) control protocol. If we simply forwarded the announce-listen control traffic across the bottleneck, the link would quickly saturate as the number of group members in either session grew. However, we prevent this condition by placing SSGs that limit the rate of the control traffic across the bottleneck link in an intelligent manner to meet the bit-rate constraints.

Using our state table model of an announce-listen protocol, the goal of the SSG is to minimize the number of inconsistencies in the protocol state table entries on both sides of the bottleneck link. More specifically, the goal of the SSG is to minimize the state update and refresh delay, subject to the input/output bit-rate constraints and potential packet loss between the servent and client. The SSG achieves this by building a cache of the *(key,value)* table and transmitting the contents of the table in a manner that obeys the bit-rate limits. The policy that determines how the table values are transmitted is the critical performance factor for the SSG.

To perform efficient rate control for announce-listen traffic, the SSG must take into account the semantics of the underlying communication. Indiscriminately dropping ASCP messages could result in potentially long latencies in the state table updates, thereby negatively impacting the operation of the system. For example, dropping new servent announcements would prevent a client from learning about the existence of a servent handling its service request. Consequently, we must be careful in the rate-limiting algorithm to favor event-triggered updates as opposed to background refreshes.

The structure of the AS1 SSG is as follows. The SSG *output channel* (OC) manages communication between servents and clients and is responsible for the rate-limiting of the ASCP control traffic. The creation and persistence of the output channel structure is driven by the ASCP announcements from the client, i.e., an output channel is a soft state structure.

The first level of rate-limiting that the OC performs is to filter out all the servent announcements that are of no interest to clients receiving the transmissions of the OC. This is easily performed since the client-side ASCP announcements define the set of SIDs that an output channel's associated clients are interested in. This in turn exactly defines the set of servents whose control traffic should be forwarded on the output channel. Since the ASCP announcements are soft-state, the resulting filter is a soft-state structure as well.

Once the OC has determined which state announcements should potentially be output from the SSG, it must output these announcements in a manner that obeys the bit-rate constraints. In the event the these constraints force some announcements to be dropped at the SSG, the OC must select for transmission announcements that maximize the state update rate of the clients.

To perform this selection, the OC maintains an announce-listen state table, in our case the ASCP state table. This table is maintained according to all the timer/refresh rules of the announce-listen protocol whose state it represents. Thus, the table represents a cache of the current state of the protocol. Using this state table, announcements are classified into two groups: *new* announcements, representing state that is not in the cache, and *refresh* announcements, representing state that is already in the cache. In ASCP, this classification is particularly easy since a servent transmits only one type of announcement, which specifies the SID that it is associated with. Therefore, the announcement classification is performed by a simple lookup into the state table keyed by the SID.

Once announcements have been classified, the OC performs rate-control using a leaky bucket mechanism. Each channel has associated with it two token buckets, or queues, NEW and REFRESH, for new and refresh announcements respectively. The queues are then serviced, i.e., announcements in the queue transmitted, at the token bucket rate.

The partition of the aggregate channel bandwidth among the NEW and REFRESH queues directly affects the tradeoff between update and refresh latencies. Allocating a higher bit-rate to the NEW bucket decreases the latency for new state, but increases the latency for refreshes. Alternatively, allocating a higher bit-rate to the REFRESH bucket maintains a high refresh rate, but delays the update of new state at the client. In our current design, we statically allocate 75% of the output rate to background refreshes and 25% to new state announcements. However, in the future we plan to leverage the schemes developed by Sharma *et al.* [29], which adapt the update timers in soft-state protocols based on channel topology models, to explore methods for dynamically varying the update and refresh rate allocation.

In summary, the SSG serves both as a point-of-contact for rate-limited control traffic and as a rendezvous point for client that do not have multicast service. Despite the fact that the SSG offers the appearance of a centralized control model, its soft-state structure enables trivial regeneration and replication, thereby avoiding a "single point of failure" design, and maintaining the overall robustness of the AS1 architecture.

3 The MeGa Active Service

Media gateways [3] are application-level agents that transparently bridge two MBone RTP sessions and process the media streams between the sessions. Having defined the AS1 re-usable framework, we now describe how this framework is specialized to the deployment of media gateways in the Media Gateway active service (MeGa).

In our framework, a media gateway is cast as a servent. We implemented MeGa to run on top of AS1 and thus provide a robust and scalable architecture for media gateway deployment that serves as a fully operational and deployed "proof of concept" for the AS1 design. Moreover, the use of RTP as the media transport protocol at the gateway guarantees the seamless integration of MeGa into the Internet multimedia infrastructure.

In this section, we describe the design of the MeGa service within the AS1 framework and focus on the service-specific components of the AS1 framework: the SID and SSD specifications for ASCP, and the service control protocol.

3.1 SID Naming

Section 2.3.1 described how the ASCP SID field is used to determine whether or not a servent should be instantiated in response to a client message. Thus, the SID naming scheme fully determines the number of servents deployed in response to a given number of client requests.

In MeGa, we use SID naming to implement a gateway "deployment policy". Specifically, a client may request that the output address of the gateway be unicast or multicast. A gateway is deployed on behalf of every unicast request, while a single gateway per session is shared among all clients requesting a multicast output address. In other words there exists only one SID for each session while for unicast requests there exists a unique SID for every client. This leads to the following specification of the SID in MeGa. For multicast requests, the SID is:

```
sspec:media
```

while for unicast requests the SID is

```
sspec:media:localaddr/rport
```

where `sspec` is the unique session name given by the session creator (e.g., the `o=` field in an SDP [18] announcement), `media` is the type of media, `localaddr` is the local IP address of the client host, and `rport` is the port on which it will receive the data.

Even though the design of ASCP is independent of the MeGa goals, through appropriate naming of the service instances, the one-to-one relationship between SIDs and servents enables us to establish a MeGa-specific servent deployment policy.

3.2 SSD Data

The role of the ASCP SSD field is to exchange initial configuration and rendezvous information between the client and servent. In MeGa, the initial servent configuration information consists of the "global" session address information, so that the gateways can join the requested session. The gateways then transmit the transcoded version of the session on a local address, which depending on the service request, might be either a multicast or unicast address. In either case, the gateway must notify the client of this address so that the client can receive session data. This exchange of global and local session information between client and gateway is performed using the SSD field of the ASCP announcements. Specifically, the MeGa client transmits the global session address in its SSD field, while the gateway SSD field contains its local transmission address.

Figure 8 illustrates a specific example of how ASCP operates in MeGa. The figure details the exchange of MeGa-specific information contained in the SSD fields to emphasize how the MeGa clients and media gateways rendezvous. The figure shows four MeGa clients: three video clients, labeled *vic*, and one audio client, labeled *vat*. The clients announce their interest in MBone sessions

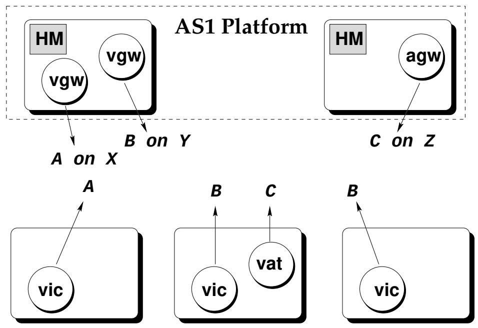

Figure 8: ASCP in MeGa. MeGa clients announce interest in sessions while media gateways announce the session for which they are a gateway along with their local transmission address. Clients "join" the global session by reconfiguring themselves to listen to the appropriate local address.

(in the ASCP SSD field). These announcements are received by the HMs who then can look in their table and see if there already exists a gateway for the given session. If no such gateway exists, the HM launches the gateway, and configures it to listen on the appropriate session address given in the client SSD field.

The next stage of the gateway launch is the selection of an output transmission address for the transcoded version of the session. In the event that the client request specified a unicast address, this involves little more than a unique port selection. However, if the request was for a multicast transmission address, we must select a unique multicast address. Obtaining this address is part of the larger multicast address allocation problem which is currently under review in the IETF and we intend to leverage their results when they become available. In the mean time we use an *ad hoc* solution where addresses are chosen randomly from a fixed block of addresses and rely on the servent announcements to detect and correct collisions. In our current prototype we have co-located the address allocation mechanism at the HM, though in the future these mechanisms can be separated. Thus, the HM launches the gateway and notifies it of the output transmission address.

Once the gateway has obtained a local transmission address, it announces this address in its SSD field. Since the MeGa SIDs are chosen so that a client and gateway SID match if there is a match between the session the gateway is handling and the session the client is interested in, the client can detect that an appropriate gateway already exists and use it to receive transcoded transmissions of the session. Thus, in the figure, the clients for sessions A, B, and C configure themselves to join group addresses X, Y, and Z, respectively, thereby completing the rendezvous between the MeGa clients and gateways.

3.3 Service Control

Service control protocols are embedded in the servents to enable the clients to implement service-specific control of the servents. In MeGa, the principal goal is the allocation of constrained link bandwidth among the media sources from a gateway to a client. Since video streams dominate bandwidth consumption on the link, we focus on a control protocol for video gateways. In the MeGa architecture, this control is carried out by the Scalable Consensus-based Bandwidth Allocation (SCUBA) protocol. Reference [2] provides a comprehensive description of the protocol. In this section, we briefly describe its operation and focus on how it relates to gateway

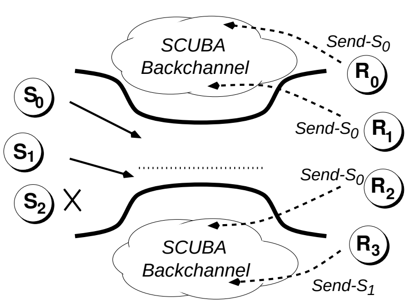

Figure 9: Receiver-driven dynamic allocation: sources dynamically adjust their transmission rate in response to receiver interest.

control in MeGa.

The basic premise of SCUBA is to reflect receiver interest back to the sources in a multicast session using a scalable control protocol. In current MBone sessions, bandwidth is allocated in a fixed fashion. Each receiver transmits at some fixed rate, where the rates are chosen either manually or through sender-based adaptation in a fashion; in either case, an equal amount of bandwidth is typically allocated to each source. Clearly, this approach is suboptimal if all sources are not equally important to the receivers. But, by integrating receiver feedback into the adaptation process, we can weight each source's transmission rate in proportion to receiver interest. This approach is illustrated in Figure 9, where receivers generate feedback that causes source S_0 to transmit at a higher rate than source S_1; moreover, because there is no interest in source S_2, its transmission is disabled entirely.

As with ASCP, SCUBA is an announce-listen protocol using only soft state. Receiver interest is expressed back to the sources in periodic, self-descriptive announcements. As a result, sources and receivers can join and leave the session at will without impacting other session members. No individual piece of the state maintained by the source is critical to the correct execution of the algorithm since all state eventually times out (or is explicitly replaced) and must be refreshed by receiver reports. As in ASCP, failure recovery is built into the protocol; we need no further mechanism to handle network partitions, host failures, and so forth. Finally, SCUBA control messages are idempotent — each message supersedes all previously sent messages — further enhancing the protocol's scalability and its resilience to packet loss.

SCUBA was designed for both session-wide deployment and media gateway control. Because we can model each media source as originating from the servent, we can run SCUBA locally between the receivers and the transcoders in the media gateways in order to partition the managed link bandwidth among the sources being transcoded by the gateway. By running SCUBA between the low bandwidth linked receivers and the gateway, scarce bottleneck bandwidth can be dynamically apportioned in an intelligent manner among the transcoders. In this way SCUBA provides a robust and distributed control mechanism for the gateway free from the vulnerabilities of centralized control.

4 Implementation Status

The AS1 framework and the MeGa service have been fully implemented and in regular use on the UC Berkeley campus for several months. The service is deployed on the Berkeley Network of Workstations (NOW) [4] using the host manager deployment algorithm described in Section 2.3.3. The only unimplemented portion of the design described above is service composition and the use of an automatic service location mechanism as detailed in Section 2.2.

In its current form the MeGa architecture contains four clients and four matching gateways for each of the following media: video, audio, whiteboard and SDP.

The audio and video gateways are implemented using the RTP gateway architecture described in [3]. The SDP gateway is implemented as a reflector. The requirements of whiteboard gateways differ significantly from the other three due to the fact that whiteboard data must be transmitted reliably, as opposed to the unreliable transmission requirements of audio, video and SDP data. In our current prototype we focussed only on the design and implementation on "stateless" audio and video media gateways. As a result, our whiteboard gateway is implemented a simple reflector. However, in the future we intend to leverage the initial efforts by Chawathe *et al.* [11] to develop an architecture for reliable real-time multicast gateways as a component of our service.

ASCP is as a string-based protocol. Our choice of a string format over a binary format was motivated by several factors. String-based protocols offer a much greater degree of flexibility in message construction. Messages can be read and written using common text-based tools which significantly reduces the amount of effort required for prototype development (e.g., one way to "send" a message is to simply type the message to the receiver's port using the *telnet* protocol). Many times the protocol information is fundamentally text-based, e.g., URL's, user names, and free text. As a result, the overhead of the text information dominates the savings that would be gained by compacting the messages to a binary packet format. Many such protocols have been designed, including HTTP, SDP, RTSP, and SIP.

The MeGa SSD field of the ASCP announcements is derived from SDP due to the close relationship between an SDP session announcements and the information required by the clients and gateways in MeGa. Thus we avoided having to design an entirely new message format and could leverage our existing SDP parser for message parsing.

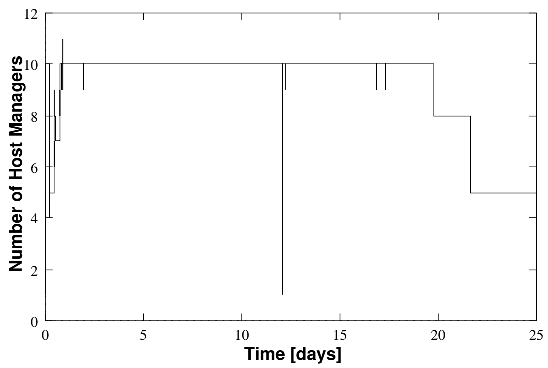

Figure 10: Host manager population changes on the Berkeley Network of Workstations over a 25 day period.

The host manager deployment algorithm has proven to be extremely robust in the presence of pathological operating conditions on the Berkeley NOW. The NOW consists of 114 UltraSparcs used by several departments on the Berkeley campus. Machines are rebooted irregularly and without notice. We deployed 10 HMs on a set of 40 machines in this cluster. The system stayed up for approximately six weeks providing robust and reliable user service until we brought it down for an upgrade. The vast majority of the time, the system was stable. Despite the occasions when machines running HMs were rebooted, the population adapted flawlessly.

Figure 10 illustrates the adaptation over the 25 days of this period (after this period there were no changes in the number of host managers). The figure plots a time series of the number of host managers present in the system. Throughout the 25 days, the system survived many system reboots and utilized 17 of the 40 hosts at one point or another. Two specific points of interest demonstrate the resilience of our system. First, on day 12 the entire cluster was rebooted. As illustrated by the downward "spike" in the graph, the population was able to survive due to the fact that the interval over which the cluster was rebooted was sufficiently long so that the randomization in the selection of nodes on which HMs replicate themselves enabled the HM population to replicate itself on the newly rebooted machines before the old population was entirely terminated — thereby ensuring the "survival" of the HM population. The second point of interest is the final configuration, in which we notice the presence of five HMs in the system. It turns out that the NOW system administrators upgraded the system's security on all but five machines, and subsequently gradually rebooted machines in the cluster. This upgrade prevented the HMs from replicating to reach the target number of 10. However, the resilience of the system was demonstrated in that the HMs attempts to replicate on random nodes of the cluster caused the system to eventually populate exactly those five machines that the system administrators did not upgrade!

5 Related Work

As we stated in the introduction, our active service framework represents an attempt to provide "active" functionality within a restricted, yet useful, subspace of active networks goals — deployment of application-level computation — while maintaining compatibility with the current Internet. In this section we present current research in active networks and describe how it relates to active services.

Govindan *et al.* [16] give a high level description of a framework for application-level active services. The report outlines an architecture for the active nodes in their network and discusses some of the research issues involved, including service deployment and the design of the service platform.

In contrast to our focus on application-level deployment and fault tolerance, most research on active networks addresses support for the more ambitious goal of enabling efficient and safe computation on arbitrary nodes at the *network* layer.

The SwitchWare project [17] is developing an architecture for programmable switches and routers. SwitchWare takes a language-based approach towards exploring the most extreme version of active networks where each packet executes a program. In addition to "active packets," the SwitchWare architecture defines middleware "switchlets" that provide support for relatively simple and lightweight packets to embody complex functionality. Alexander *et al.* [1] describe an implementation of an "active bridge" implemented entirely by switchlets running within the SwitchWare architecture. A related effort is the BBN "smart packets" and "active router" projects [20]

Bhattacharjee *et al.* [9] describe an active networks architecture for dealing with congestion in the network. They detail the use of "active processors" —- software modules that implement application specific processing on a packet-level basis. These packets are labeled and are dropped in the face of congestion according to a "unit-level drop" function that enables the user to specify the granularity of adaptation. One of their examples is the use of an active processor for MPEG streams to control packet loss in the face of congestion. This problem is obviously very similar to that addressed by MeGa and, in a sense, media gateways are active processors. The main difference is that this approach comes from the network up, while MeGa addresses the problem from the application-level down. While the former achieves increased generality, it does so by sacrificing the ability to leverage useful information from higher-level protocols such as SCUBA.

The NetScript [35] project's goal is the design of a common language and execution environment to provide a universal abstraction of a programmable networking environment. NetScript is orthogonal to and complementary to our work and we foresee a possibility of leveraging it in our active service framework when it becomes more refined.

Finally, Wetherall and Tennenhouse describe a mechanism for deploying computation in the network using an new option in the IP header: the ACTIVE IP option [34] in conjunction with embedding the actual code, or "capsules" [31], in the network-level packet header. Similar to our goals, this approach is motivated in part by the goal of compatibility with today's Internet.

6 Summary

In this paper we described Active Services, an architecture for deployment of application-level computation within the network. Our active service architecture draws upon three important protocol building blocks — announce-listen communication, soft-state, and multicast damping — which together yield a particularly robust and flexible design. To demonstrate the efficacy of our architecture, we implemented an active service for media gateways called MeGa. MeGa incorporates the core components of an active service and serves as a fully functional and deployed "proof of concept" for our work. Active Services address an important subset of the problems targeted by the active networking initiative while preserving compatibility with the current Internet infrastructure.

7 Acknowledgments

We thank Prof. James Landay for providing his CSCW course at UC Berkeley as an invaluable testbed for MeGa in a production environment. During the course, Robert Wilensky, Teck-Lee Tung, and Angela Schuett provided valuable user feedback. We thank our colleagues Hari Balakrishnan, Venkat Padmanabhan, Mark Stemm, and Todd Hodes for the input throughout the development of work in this paper. David Culler, Rich Martin, and Eric Fraser helped in deploying AS1 and MeGa on the Berkeley NOW. Finally, we thank the anonymous reviewers for their comments.

References

[1] ALEXANDER, D. S., SHAW, M., NETTLES, S. M., AND SMITH, J. M. Active bridging. In *Proceedings of SIGCOMM'97* (Cannes, France, Sep 1997), ACM.

[2] AMIR, E., MCCANNE, S., AND KATZ, R. Receiver-driven bandwidth adaptation for light-weight sessions. In *Proceedings of ACM Multimedia '97* (Nov. 1997), ACM.

[3] AMIR, E., MCCANNE, S., AND ZHANG, H. An application-level video gateway. In *Proceedings of ACM Multimedia '95* (Nov. 1995), ACM.

[4] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A case for networks of workstations: NOW. *IEEE Micro* (Feb. 1995).

[5] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison-Wesley, 1996.

[6] ARPANET WORKING GROUP REQUESTS FOR COMMENT, DDN NETWORK INFORMATION CENTER. *Dynamic Host Configuration Protocol (DHCP)*. SRI International, Menlo Park, CA, October 1993. RFC-1541.

[7] ARPANET WORKING GROUP REQUESTS FOR COMMENT, DDN NETWORK INFORMATION CENTER. *Service Location Protocol*. SRI International, Menlo Park, CA, June 1997. RFC-2165.

[8] BALAKRISHNAN, H., SESHAN, S., AMIR, E., AND KATZ, R. H. Improving TCP/IP performance over wireless networks. In *Proceedings of 1st ACM Conf. on Mobile Computing and Networking (MOBICOM)* (Berkeley, CA, November 1995), ACM.

[9] BHATTACHARJEE, S., CALVERT, K. L., AND ZEGURA, E. On active networking and congestion. Technical Report GUT-CC-96/02, College of Computing, Georgia Institute of Technology, Atlanta GA, 1996.

[10] BORENSTEIN, N. E-mail with a mind of its own: The Safe-Tcl language for enabled mail. In *Proceedings of IFIP International Conference* (Barcelona, Spain, 1994).

[11] CHAWATHE, Y., FINK, S., MCCANNE, S., AND BREWER, E. A proxy architecture for reliable multicast in heterogeneous environments. In *Proceedings of ACM Multimedia '98* (Sept. 1998), ACM. *To appear*.

[12] CHESSON, G. XTP/protocol engine design. In *Proceedings of the IFIP WG6.1/6.4 Workshop* (Rüschlikon, May 1989).

[13] CLARK, D. D. The design philosophy of the DARPA Internet protocols. In *Proceedings of SIGCOMM '88* (Stanford, CA, Aug. 1988), ACM.

[14] FENNER, W. *Internet Group Management Protocol, Version 2*. Internet Engineering Task Force, Inter-Domain Multicast Routing Working Group, Feb 1996. Internet Draft (work in progress).

[15] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C.-G., AND ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM '95* (Boston, MA, Sept. 1995), ACM.

[16] GOVINDAN, R., ALAETTINOĞLO, C., AND ESTRIN, D. A framework for active distributed services. Technical Report 98-669, Information Sciences Institute, University of Southern California, Los Angeles CA, 1998.

[17] GUNTER, C. A., NETTLES, S. M., AND SMITH, J. M. The SwitchWare active network architecture, Nov 1997. White paper available at http://www.cis.upenn.edu/˜ switchware.

[18] HANDLEY, M., AND JACOBSON, V. *SDP: Session Directory Protocol*. Internet Draft, Mar 26, 1997.

[19] HODES, T. D., KATZ, R. H., SERVAN-SCHREIBER, E., AND ROWE, L. A. Composable ad-hoc mobile services for universal interaction. In *Proceedings of The Third ACM/IEEE Interattional Conference on Mobile Computing (MOBICOM)* (Budapest, Hungary, Septmber 1997).

[20] JACKSON, A. W., AND PARTRIDGE, C. Smart packets, March 1997. Slides from 2nd Active Nets Workshop[1].

[21] JACOBSON, V. SIGCOMM '94 Tutorial: Multimedia conferencing on the Internet, Aug. 1994.

[22] KLEINROCK, L. Nomadic computing, Nov 1995. Keynote Address: International Conf. on Mobile Computing and Networking (MOBICOM).

[23] MCCANNE, S., BREWER, E. A., KATZ, R. H., ROWE, L., AMIR, E., ET AL. Toward a common infrastructure for multimedia-networking middleware. In *Proceedings of the Fifth International Workshop on Network and OS Support for Digital Audio and Video (NOSSDAV)* (St. Louis, Missouri, May 1997).

[24] MCCANNE, S., AND FLOYD, S. *The UCB/LBNL Network Simulator*. University of California, Berkeley. Software online[2].

[25] NONNENMACHER, J., AND BIERSACK, E. Optimal multicast feedback. In *Proceedings of IEEE INFOCOMM 98* (April 1998).

[26] OUSTERHOUT, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[27] OUSTERHOUT, J. K. Scripting: Higher level programming for the 21st century. *IEEE Computer* (Mar. 1998).

[28] ROSSUM, G. V. Python tutorial.

[29] SHARMA, P., ESTRIN, D., FLOYD, S., AND JACOBSON, V. Scalable timers for soft state protocol. In *Proceedings of IEEE INFOCOMM 97* (April 1997).

[30] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A survey of active network research. *IEEE Communications Magazine 35* (Jan 1997), 80–86.

[31] TENNENHOUSE, D. L., AND WETHERALL, D. J. Towards an active network architecture. *Computer Communication Review 26*, 2 (Apr. 1996), 5–18.

[32] TURLETTI, T., AND BOLOT, J.-C. Issues with multicast video distribution in heterogeneous packet networks. In *Proceedings of the Sixth International Workshop on Packet Video* (Portland, OR, Sept. 1994).

[33] WETHERALL, D., AND LINDBLAD, C. J. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop* (Ontario, Canada, July 1995).

[34] WETHERALL, D. J., AND TENNENHOUSE, D. L. The ACTIVE IP option. In *Proceedings of the 7th ACM SIGOPS European Workshop* (Connemara, Ireland, Sep 1996), ACM.

[35] YEMINI, Y., AND DA SILVA, S. Towards programmable networks. In *Proceedings of IFIP/IEEE International Workshop on Distributed Systems, Operation and Management* (L'Aquila, Italy, Oct 1996), IEEE.

[1] available at http://www.net-tech.bbn.com/smtpkts/baltimore/index.htm

[2] http://www-mash.cs.berkeley.edu/ns/