Congestion Control and Caching in CANES*

Samrat Bhattacharjee, Kenneth L. Calvert, Ellen W. Zegura
Networking and Telecommunications Group, College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{bobby,calvert,ewz}@cc.gatech.edu

Abstract

Active networks provide a meta-level *interface that can be* programmed *to customize the network's behavior on a per-user basis. In this paper, we present some applications of active networking developed by the* CANES *active networking research group at Georgia Tech.*

1 Introduction

Active networks provide a programmable platform on which network services can be defined or altered by injecting code or other information into the nodes of the network. With active networking, the network is no longer viewed as a passive mover of bits, but rather as a more general computation engine: information injected into the network may be modified, stored, or redirected as it is being transported [1]. Active networks support dynamic (i.e., on-the-fly) control of the network's behavior. Such dynamic control is potentially useful on multiple levels.

On the *global* level, such changes are motivated by the need to reduce the time required to develop and deploy new network services. The shared infrastructure of the network presently evolves at a much slower rate than other computing technology. For example, the basic concepts and algorithms of IP multicast have been known for a decade, but it is still not a "standard" service of the network infrastructure, in spite of its demonstrated utility.

On the other hand, *local* control of network behavior could enable users to create services tailored to their particular applications. While the traditional interface to the network service hides many details (e.g., topology) of the underlying network from the user for the sake of simplicity and scalability, active networks offer a more general interface. This allows network knowledge —such as the impending onset of congestion— to come together with application knowledge —such as the semantics of data units and their interdependencies— in a timely way, improving the overall utility of the service to the user. Obviously, such a capability opens up many exciting possibilities. Active networks can utilize information that is only available *inside* the network [2]. In general, applications that can benefit from such knowledge permit innovative active networking solutions. In this paper, we present a survey of applications being investigated by the CANES (Composable Active Network Elements[1]) project at Georgia Tech., that benefit from active networking. The applications that we present in detail are (i) congestion control, and (ii) caching in wide area networks. During congestion, the network knows *where* the congestion occurs. However, the source of the data usually knows best *what* to do with its data if it encounters congestion. In Section 2, we show how, using active networking, this network and end-system knowledge can be brought together to form effective application-specific congestion control schemes. Section 3 is a survey of our work on wide-area caches, and how active networking can be used to reduce access latencies in a wide-area network using relatively small caches. We present a summary of our results in Section 4.

2 Application-Specific Congestion Control

Despite advances in techniques to provide hard guarantees on quality of service, there will always be applications that prefer to use a best-effort service. These applications will dynamically *adapt* their rate to match available network bandwidth, as opposed to reserving bandwidth in advance. This expectation is based on the fact that the network will at times reject requests for reserved bandwidth and applications will have to use a non-reserved service or go away. A guaranteed service is also likely to be more expensive. Further, the recent advances in understanding the self-similar nature of traffic [3, 4, 5] indicate that effective congestion control is likely to require longer time scale methods (e.g., source adaptation and admission control) rather than shorter time scale methods (e.g., adding more buffering).

At the same time, however, it must be noted that the sender-adaptation model [6], which has worked so well in the Internet, presents well-known challenges. The first is the time interval required for the sender to

*This work was supported by DARPA under contract number under contract number N66001-97-C-8512. The work of Ellen W. Zegura was supported in part by NSF Careers Award MIP-9502669. This summarizes work originally appearing in HPN'97, and in Infocom'98

[1] http://www.cc.gatech.edu/projects/canes

detect congestion, adapt to bring losses under control, and have the controlled-loss data propagate to the receiver. During this interval, the receiver experiences uncontrolled loss, resulting in a reduction in quality of service that "magnifies" the actual bandwidth reduction. (For example, if a portion of each of several application data units is lost, each entire data unit may become useless to the receiver.) As transmission and application bandwidths increase, this problem is exacerbated because propagation delays remain constant.

The other well-known challenge of sender adaptation is detecting an *increase* in available bandwidth. This problem, which is worse for continuous-media applications, arises in traditional best-effort networks because loss is the only mechanism for determining available bandwidth. Thus, for example, if a sender adapts to congestion by changing to a lossier encoding, it must detect the easing of congestion by periodically reducing compression and waiting to see whether losses ensue. In the case of long-lived congestion, this dooms the receiver to periodic episodes of uncontrolled loss.

We claim that the best-effort service provided to adaptive applications can be enhanced by allowing applications some control over the way their packets are processed in network switches when they encounter congestion. Instead of applying "one size fits all" congestion reduction techniques, mechanisms can be placed in the network to allow packet processing —e.g. discarding or transforming— to proceed according to advice supplied by the application. The observation is that the application knows *how* to adapt to congestion, while the network knows *when and where* adaptation is needed. In essence, the adaptation that the sender would have applied can be performed at the point of congestion. This solves both of the problems noted above.

The approach we propose allows for variation in the treatment among the packets of a *single flow*. Whereas the "best-effort" service of IP treats all datagrams uniformly, and other proposals for enhanced services allow for discrimination between flows, we consider the benefits of distinctions among the different packets of a flow. Such a capability helps applications such as MPEG [7], which may not require total reliability but which benefit from "better effort" by the network on behalf of certain data units.

2.1 Operating Model

A *flow*, for our purposes, is a sequence of packets all having the same source and destination, from the point of view of a node somewhere in the network. Thus a flow might consist of packets traveling between a single pair of endpoints, or it might be the aggregation of a set of lower-level flows. We assume that a flow is identified by a label of some kind in the network protocol header.

Generically, programmable congestion control operates as follows: *Based on triggers that indicate congestion control should take place, flow state is examined for advice about how to reduce quantity of data.*

The important components of this generic model are the *triggers* responsible for initiating congestion control, the *flow state* that contains the specific advice for this flow, and the *reduction techniques* defined by the network and made available to the users. An important feature of this model is its consistency with traditional best-effort service. That is, a flow provides *advice* about what to do with its data. The network node is not required to take the advice, and may apply generic bandwidth reduction techniques.

Such transformations will be application-specific and computationally complex (not to mention the significant issues they raise regarding interaction with end-to-end mechanisms such as error detection and sequence numbering). We therefore focus on the special case of *intelligent discard* of data. Using a natural extension of packet-level discard [8], we allow applications to define units based on application semantics, with aim of discarding the entire unit if any portion must be discarded. It should be noted that for this scheme to be most effective, buffering is required to collect application data units to ensure that the "tail" will not be dropped. Further, the end-to-end reliability mechanisms (if any) affect the optimal definition of a unit. The most benefit derives from dropping units that are equivalent to the reliable retransmission units; dropping a unit that is smaller or larger may result in unnecessary retransmission.

Given that bandwidth reduction will occur by discarding units, a question arises as to which unit (within a flow) to discard. In the most simple case, there is no choice: when the congestion indication trigger occurs, a fixed unit (typically the one currently being processed) is subject to discard. More efficient network behavior is possible, however, if we expand the set of data from which we choose to discard. We consider congestion control advice that indicates how to make such a choice. One can think of this advice as indicating priority or some other policy by which to discriminate across data in the same flow. Making use of this advice clearly requires that the network node have access to a collection of data within a single flow. Thus, these mechanisms will involve storing and manipulating flow data before it leaves the node, e.g., while sitting in a per-flow queue from which packets are periodically selected for output by a scheduling mechanism.

2.2 Example Application and Mechanisms

To illustrate specific mechanisms and evaluate performance, we focus on the use of congestion control advice to improve the best-effort service provided to MPEG. For our purposes, the important feature of an MPEG stream is that it consists of a sequence of frames of three types: I, P and B. Coding dependen-

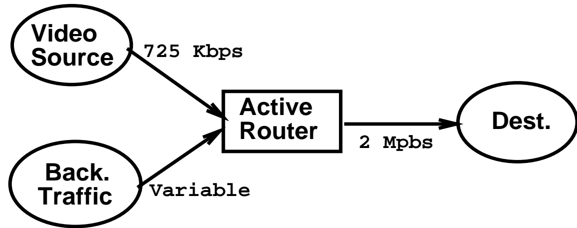

Figure 1: Emulation topology

cies exist between the frames, causing P and B-frames to possibly require other frames in order to be properly decoded. Each I-frame plus the following P and B frames forms a group of pictures (GOP), which can be decoded independently of the other frames. The specific congestion control mechanisms we evaluate are as follows:

The **Partial Packet Discard** mechanism is for baseline comparisons; it defines each IP packet to be a unit. Packets are discarded if they cannot be buffered in the output queue.

The **Static Priority** mechanism is able to distinguish packets carrying I-frames (I-packets) from other packets, and upon congestion, discards all other P- and B-packets in its queue before discarding an I-packet. Thus, this scheme provides static total priority to I-packets.

The active mechanism we consider identifies dependencies between units. Our **Group of Picture (GOP) Level Discard** maintains state about the type of frame discarded. In case an I frame has been discarded, we discard the corresponding P and B frames as well. Thus, the GOP discard recognizes both the priority of each packet and the application-level frame boundaries.

2.3 Results

Experimental Setup The experimental topology is shown in Figure 1. The "video source" source generated MPEG traffic at a given rate, and the "back. traffic" source generates trace driven background traffic at specified rates. The bandwidth on the link from the active router to the destination is constrained to 2 Mbps; thus any combination of source and background traffic that exceeds 2 Mbps will result in congestion into the destination. In this experiment, the active router had a 8K total buffer.

In this experiment, the source generates a 30 frames-per-second MPEG stream of average rate 725 Kbps. We vary the average rate at which the background traffic is generated from 1 Mbps to 2Mbps.

In order to evaluate the application-specific congestion control schemes, we consider two metrics.

- **Fraction of I Frames received.** The fraction of I frames that are received provide a good measure of the quality of the resultant MPEG stream. As our results have shown, this easily measured metric is a good approximation of the actual SNR of a stream [9].

- **Fraction of bytes that were eventually discarded at the receiver.** Under congestion, it is possible for partial MPEG frames to be received since part of the frame that was being carried in another IP packet was discarded. Also, due to the inter-dependent frame structure of MPEG, frames can only be decoded if all other frames that they depend on have also been received. We measure the fraction of data that is received but cannot be used due to incomplete frames, or missing related frames.

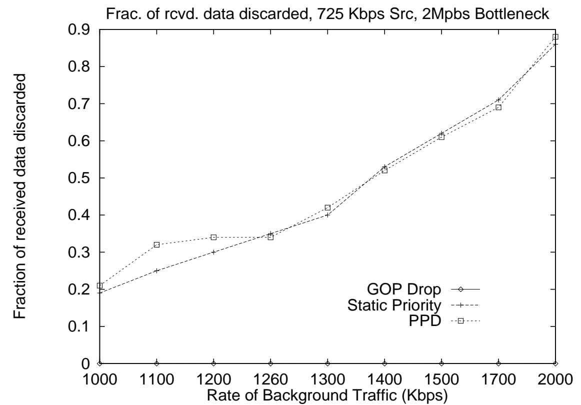

Figure 2: Fraction of received data discarded

Fraction of Received Data Discarded Figure 2 shows that as the average background traffic rate is increased, for the Static Priority and PPD schemes the amount of useless data received increases. Under heavy congestion, nearly 90% of the data received is discarded. Note that 30–40% of the received data is discarded by these non-active mechanisms when the background traffic varies between average rates 1200–1300 Kbps. This range of background traffic corresponds to the case when the the average incoming traffic rate at the router is about 96–101% of the outgoing link capacity.

Not only does the data discarded at the receiver result in the inevitable poor quality of the received video stream, it also results in a large fraction the channel allocated to the receiver being effectively wasted. The active mechanism (GOP discard) achieves 100% channel utilization — none of the data received is discarded regardless of the rate of the background traffic.

I-frames received Figure 3 shows the fraction of I-frames sent that are received without error at the receiver. As the background traffic rate increases, the fraction of I-frames received drops *steeply* for the non-active mechanisms (Static Priority and PPD). The fraction of I-frames received under the active mechanism (GOP) also decreases, but the degradation is far

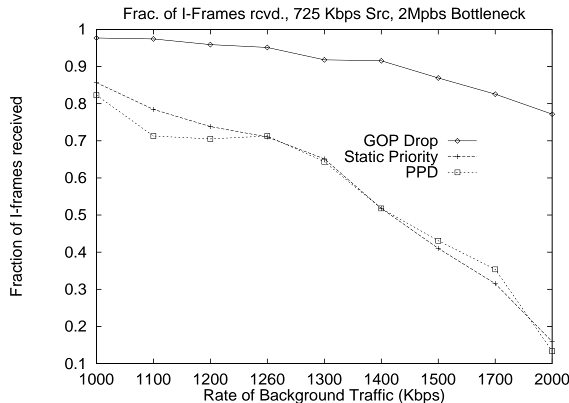

Figure 3: Fraction of I-frames received

more graceful. If we consider the background rates between 1200–1300 Kbps (96–101% of outgoing channel capacity), the non-active mechanisms deliver between 65–75% of I-frames. Under similar background traffic rates, the active mechanism delivers 94–98% of all the I-frames.

More results, including multi-hop scenarios, more active mechanisms, destination feedback, and detailed evaluation of individual streams can be found in [9].

3 Self-Organizing Wide-Area Caches

Traditional approaches to network caching place large caches at specific points in the network. In contrast, we consider networks in which relatively small caches are placed at every node. As a response message moves through the network, each node decides whether or not to store the object. Effective use of a large number of small caches is a non-trivial problem: unless they are effectively organized, only a small number of unique items will be cached (as the caches are much smaller), but these cached items will be replicated at many locations throughout the network. Thus, accesses to the few cached objects will exhibit low latencies, but overall average latency will not decrease appreciably. However, if objects are cached too sparsely, then the latency again does not decrease and caching provides little benefit.

In this section, we show how such small caches can be organized using "active" schemes, and present simulation results of their performance in comparison to existing caching schemes.

3.1 Self-Organizing Caches

We describe our self-organizing schemes as if every node of the network caches objects, but all that is required is that caches be reasonably uniformly distributed. Thus, these schemes obviate the need to decide *where* to place caches which can be a critical decision for traditional caching mechanisms.

In what follows, the network is considered to be a collection of *domains*, each of which is represented as a graph of switching nodes connected by links. Domains are of two types, *transit*, which (as their name implies) carry transit traffic, and *stub*, through which only packets addressed to or from some node in the domain are carried. The graph models used in our simulations are constructed using the GT-ITM Internet topology modeling package [10]. These graph models ensure that the paths along which packets travel in the simulations have the characteristics that (i) the path connecting two nodes in the same domain stays entirely within that domain, and (ii) the shortest path connecting node u in stub domain U to node v in another stub domain V goes from U through one or more transit domains to V, and does not pass through any other stub domains. Note that "nodes" in these models represent routers, and end systems are not explicitly modeled. Thus references to "servers" or "server nodes" should be interpreted as meaning nodes to which one or more servers are connected.

We assume an application in which clients request objects from servers located throughout the network. Each object is assumed to have a globally unique identifier (e.g. a hash function of a URL or message body), and to fit in a single "message". Each transaction is initiated by a *request message* sent from a client toward a server, containing the ID of one requested object. The request message travels through the network until it reaches a node where the requested object is stored, which may be a cache or the server itself. A *response message* containing the object then travels from the server (or cache) back to the originating client, completing the transaction. For simplicity, we assume messages are never lost. This simple model ignores the problem of end-to-end transfer of objects too large to fit in a single message. However, it is adequate for our purposes, namely comparison of active caching methods with traditional ones.

Our goal is to have nodes make local decisions about which objects they place in their (small) caches, in such a way that resources are used effectively overall. In particular, we wish to avoid having the same (few) objects cached at most of the nodes of the network. We describe two related approaches.

Modulo Caching. To ensure that cached objects are distributed through the network, we introduce a distance measure called cache *radius*, measured in transmission hops. The caching policy uses the radius measure as follows: on the path from the server (or cache) to the requesting client, an item is cached at nodes that are the cache radius apart. Thus, an object ends up being distributed in concentric "rings" centered on the server where it resides; the rings are separated by a number of hops equal to the cache radius. The radius is a parameter of the policy; it might be set globally, on a per-object basis, or even locally in different parts of the network. (Our simulation results assume a common global cache radius, equal to 3.)

The mechanism used to implement this policy locally at each node is a simple modulus. The response message contains a hop count that is initially set to the

object identifier modulo the radius; the count is incremented by each node through which the packet passes. When the incremented count modulo the cache radius equals zero, the object is cached at that node.

Lookaround. Network caches store relatively large objects compared to the amount of space required to store the location of an item within the network. For example, an object in a network cache can be several thousand bytes, while its location could be an IP address (4 bytes). This fact can be exploited by having a self-organizing cache dedicate some of its cache space to store *locations* of (nearby) items.

Caching nodes keep a periodically-updated list of items cached at neighbors. Logically, each node's cache is divided into "levels": level zero contains objects cached locally. Level one contains the locations of objects cached at nodes one hop away, level two contains locations of objects cached at nodes two hops away, etc. When a request message is processed, the levels are searched in sequence beginning with zero; if a hit is detected in a nearby cache, the request is simply re-routed to that node (source and destination addresses are not changed). If the information about the neighbor's cache turns out to be incorrect, the neighbor simply forwards the datagram toward the destination. Thus, the mechanism is fail-safe and backward compatible: a mix of active and non-active nodes may exist in the network, and the active cache functions may fail at any time and fall back on the regular forwarding functions. (In our simulations, we constrain the lookaround to nodes in the same domain.)

The number of levels of adjacent caching maintained and checked in this *lookaround* algorithm becomes a parameter of the policy. With this approach, even very small caches can look like "virtual" large caches. We refer to this extension of the modulo caching scheme as **Modulo Caching with Lookaround**.

3.2 Traditional Mechanisms

Our simulations compare the above self-organizing caching schemes to "traditional" caching schemes, in which each cache attempts to store each (popular) object passing through it, without any global coordination beyond the placement of the cache within the network. We consider the following placement strategies:

Cache at Transit Nodes ("Transit-Only"). Transit nodes have to be traversed for every non-local stub domain access; a large fraction of paths in the network have to go through transit routers. This ubiquity of transit nodes in network paths make them prime candidates for caches.

Cache at Stub Nodes Connected to Transit Nodes ("SCT"). Stub nodes connected to transit nodes have to be traversed in order to access the transit network. Thus, these stub nodes form good locations for network caches.

Cache at Every Node ("No AN"). We also consider an approach in which caches are located in every node (like the self-organizing schemes), but without any coordinating mechanisms enabled. This case corresponds to a "null" active function in an active network. We refer to it as **"No AN"**.

3.3 Caching Results

We compare the performance of traditional and self-organizing techniques for wide-area caching using a locally developed discrete event network simulator called AN-Sim. AN-Sim simulates an active network as described in [9], and allows for realistic models of the network topology.

We simulated many networks that differ in number of nodes, diameter, average node degree, ratio of transit nodes to stub nodes, etc. For the results that follow, a 1500 node graph was used. It had 60 transit nodes, 1440 stub nodes, and average degree 3.71 Each stub node is assumed to connect to one server, thus each graph has 1440 servers. A subset of the servers, chosen uniformly at random, are designated to be *popular servers*. The number of popular servers is nominally 300. (One experiment explores the effect of varying the number of popular servers.) There are 4 billion (2^{32}) unique objects in each simulation, the vast majority of which are not accessed. Each object is associated with a particular server, thus each server's content is unique. A subset of objects at each popular server is designated to be popular. (Unpopular servers have only unpopular items.) The number of popular objects is fixed at 48 per popular server (for a nominal total number of popular objects of 14400.)

To decide upon a query, first a client is chosen. The client picks a server, then picks an object at the server. The access patterns governing the choice of client, server and object at a server are described below. Every stub node is assumed to connect to one client. We simulate several different access patterns. In the *uniform access pattern*, a client is chosen at random. Then, a server is chosen using a Zipf distribution: the popular servers are chosen with a probability $1 - \epsilon$. All other client requests go to an unpopular server, chosen at random. If a popular server is selected, then a popular object at that server is selected 95% of the time. The remaining 5% of the accesses to a popular server select from the unpopular objects. If an unpopular server is selected, then a random object is selected.

In the *correlated access pattern*, accesses are not independent, but rather may involve the same client and server pairs. In the correlated access pattern, there are two types of accesses: initial accesses and dependent accesses. An initial access is generated using the uniform access pattern described above. Initial accesses

are separated in simulation time using an initial access inter-arrival time distribution. With a fixed correlation probability, an initial access is the "anchor" for a train of future dependent accesses that are interspersed in time with the initial access generation process. A train has an average length of 16 accesses; the time of each future dependent access is determined by an offset distribution from the time of the anchor access. A dependent access has the same client and server as the anchor access. With a fixed probability, the item selected in a dependent access is the same as the previous item in the train. Otherwise, the item is selected according to the uniform access pattern described above.

Performance Metrics We use the following metrics to evaluate the performance of network caches.

- **Round trip length (RTL).** We measure the number of hops traversed by the packets involved in a transaction. This is perhaps the simplest and most "true" measure of network cache performance.
- **Fraction of queries that generate cache hits.** After an initial startup period, the cache performance stabilizes. We measure the fraction of queries that are serviced by cache hits. Note that queries served by caches not only reduce access latencies and conserve bandwidth, but also reduce server load.

In the rest of this section, we present details of a small cross section of our experiments.

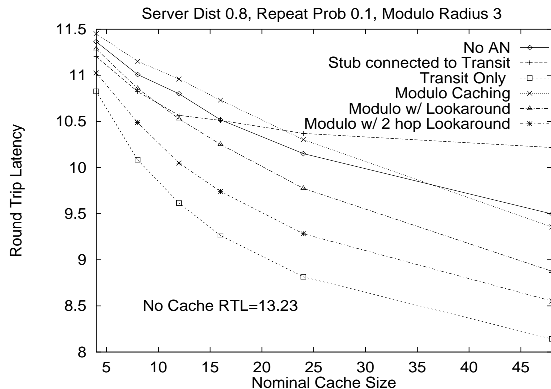

Figure 4: Latency with Low Access Correlation

Variation in Cache Size We varied the nominal cache size from 4 to 48 cache slots per interface. The Modulo and No-AN methods use nominal cache size, as these methods cache at all nodes. The corresponding numbers for Transit-Only caching are 33 to 397 slots per interface, and for Stubs-connected-to-Transit (SCT) caching, 15 to 188 slots/interface. All of the caching mechanisms, except SCT, show a smooth decrease in number of hops traversed per round trip as the cache size is increased.

In Figure 4, the probability of repeating an accesses within a set of correlated accesses is 0.1. The Modulo cache radius is fixed at three. The Lookaround schemes perform better than all but the Transit-Only caching scheme, and the performance of the two-level Lookaround scheme is within 10% of the Transit-Only scheme in all cases. It should be noted that the number of cache slots per interface for the two-level Lookaround scheme is an order of magnitude smaller than for the Transit-Only scheme. Also, the average degree of the transit nodes is much greater than the average degree of the graph. Thus the transit node caches are 25 times larger than the Modulo caches. Comparatively, the SCT caches average 4.25 times larger than the Modulo caches.

As accesses become more correlated, as shown in Figure 5 (repeat probability 0.5), the Modulo with Lookaround scheme outperform all others. Also significant in Figures 4 and 5 is the behavior of the caches in the SCT scheme. The performance improvements are negligible beyond 12 cache slots per interface, and as such this method does not scale well with increase in cache size.

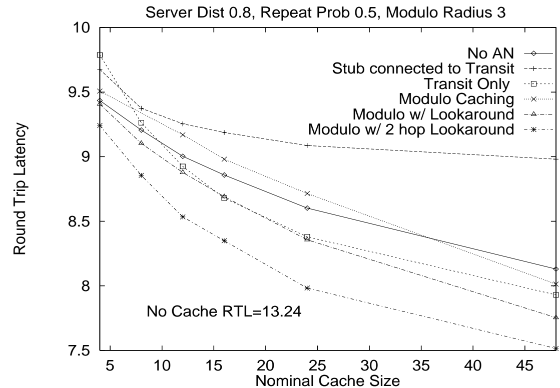

Figure 5: Latency with High Access Correlation

For the same parameters as in Figure 5, Figure 6 shows the fraction of queries that generate cache hits. Once again, all the methods except SCT show improvement with increase in cache size. The SCT method actually results in a proportionately large fraction of hits—but the large number of hops in the round trip suggests that more hits occur in the stub node to which the *server* is connected, and not the client. This is not unexpected—the *gateway* stub node connected to the transit domain for a busy server will experience a lot of traffic due to the busy server, and as such, will cache a large part of that data as well.

Variation in Server Distribution We have used a Zipf distribution to model server popularity (i.e. a

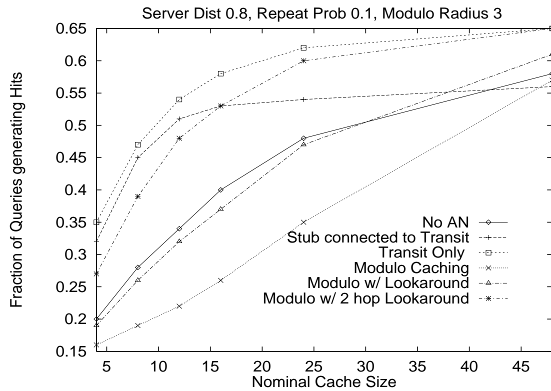

Figure 6: Cache Hits with High Access Correlation

fraction $1-\epsilon$ of the accesses are to the fraction ϵ of the nodes). However, it is not clear exactly what fraction of the nodes should be considered to be *servers*. In this experiment (Figure 7), we vary the fraction of all nodes that are servers from 0.01 to 0.5. Even when a large fraction of nodes are servers, the cache performances are not significantly affected. Thus, wide-area caching seems robust in face of widely varying server locations and distributions. It is interesting to note that round trip latencies for Transit-Only cache schemes do not improve much when the server distributions are extremely skewed, e.g. when less than 10% of the nodes are servers. The other schemes improve as the number of popular objects (which is a multiple of the number of servers) decrease, but in case of Transit-Only caches, even if all the objects are cached everywhere, the query has to reach the transit nodes before it is serviced.

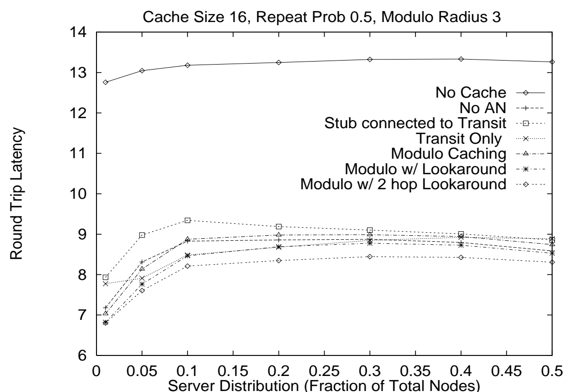

Figure 7: Latency with Varying Server Distributions

Details on these and other results, including more topologies, access schemes, and an analytic model can be found in [11].

4 Summary

Active Networking is a new area of networking research with potentially many exciting applications. In this paper, we have presented a summary of two applications of active networking developed by the CANES research group at Georgia Tech. Our experience with application-specific congestion control shows that active networking can used to provide better service when the network is congested to applications that use best-effort service. Further, the relative performance of the active congestion control schemes improves as congestion worsens. Our work on self-organizing caching mechanisms show that active networks can be used to coordinate sets of small caches in order to reduce access latencies. Our results show that active caching is beneficial across a range of network topologies and access patterns, and it is especially effective when access patterns exhibit significant locality characteristics.

References

[1] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, 1997.

[2] S. Bhattacharjee, K. Calvert, and E. Zegura, "Active networking and the end-to-end argument," in *Proceedings of ICNP'97*, 1997.

[3] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of Ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking*, pp. 1–15, February 1994.

[4] N. Likhanov, B. Tsybakov, and N. Georganas, "Analysis of an ATM buffer with self-similar (fractal) input traffic," in *IEEE Infocom '95*, 1995.

[5] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson, "Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level," in *ACM Sigcomm '95*, pp. 100–113, 1995.

[6] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM '88*, 1988.

[7] I. O. for Standardisation, "Generic coding of moving pictures and associated audio." ISO/IEC/JTC1/SC29/WG-11, March 1993.

[8] G. Armitage and K. Adans, "Packet reassembly during cell loss," *IEEE Network Magazine*, September 1993.

[9] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, "An Architecture for Active Networking," in *Proceedings of High Performance Networking 97*, 1997.

[10] K. L. Calvert, M. B. Doar, and E. W. Zegura, "Modeling Internet Topology," *IEEE Communications Magazine*, June 1997.

[11] S. Bhattacharjee, K. Calvert, and E. Zegura, "Self-organizing wide-area network caches," in *IEEE Infocom'98*, 1998.