Multicasting Multimedia Streams with Active Networks

Albert Banchs[1], Wolfgang Effelsberg[2], Christian Tschudin[3], Volker Turau[4]

TR-97-050

Abstract

Active networks allow code to be loaded dynamically into network nodes at run-time. This code can perform tasks specific to a stream of packets or even a single packet. In this paper we compare two active network architectures: the Active Network Transfer System (ANTS) and the Messenger System (M0). We have implemented a robust audio multicast protocol and a layered video multicast protocol with both active network systems. We discuss the differences of the two systems, evaluate architectural strengths and weaknesses, compare the runtime performance, and report practical experience and lessons learned.

Keywords: *Active Network, ANTS, M0, robust audio, scalable video, layered video*

[1] International Computer Science Institute, Berkeley, and Universitat Politecnica de Catalunya, Barcelona, Spain.
E-mail: banchs@icsi.berkeley.edu
[2] International Computer Science Institute, Berkeley, and University of Mannheim, Germany.
E-mail: effels@icsi.berkeley.edu
[3] International Computer Science Institute, Berkeley, and University of Zürich, Switzerland.
E-mail: tschudin@icsi.berkeley.edu
[4] International Computer Science Institute, Berkeley, and Fachhochschule Wiesbaden, Germany.
E-mail: turau@icsi.berkeley.edu

1 Introduction

Active networks allow protocol processing code to be loaded into network nodes at run-time. Based on an identifier in each packet header, a specific piece of code is invoked as the packet travels trough the node. The main advantage of active networks is the flexibility compared to traditional networks: It is very easy to implement a new protocol, to remove errors in network software, or even to provide specific processing just for the duration of one session.

In traditional networks such as the Internet or proprietary networks, application-specific processing is only done in the end systems; the transfer protocols are the same for all applications. For example, the IP protocol in the Internet is enabling many different applications, and at the transport layer there are two major protocols: TCP and UDP. Traditional applications such as e-mail, file transfer, remote login, and network news could be mapped to this small set of protocols. All transmissions were point-to-point, and consequently it made sense to do application-specific processing in the end nodes only.

The next generation of networks will have to handle a much larger variety of application traffic: audio, video, workflows, and many more. Many of them inherently require multicast. In order to support large numbers of receivers worldwide the multicast function will have to be provided in all network nodes. Such a node is already much more complex than a traditional router: it must handle packet duplication, group address management, dynamic joining and leaving of group members, and perhaps also QoS-based multicast routing. The amount of processing required for each incoming packet is increased considerably, and so is network management overhead. New signaling functionality for multimedia includes resource reservation, QoS-based routing and stream-specific packet filtering. New architectures and protocols are being designed and implemented at a much faster rate than ever. Thus an active network architecture allowing the fast deployment of new protocols and stream-specific processing in internal network nodes seems very desirable.

The proposed transition from traditional data networks characterized by the passive transport of data towards active networks allowing the network to perform computations on the data can be compared to the transition from procedural programming to object-oriented programming. While the former programming style clearly separates the notions of data and operations, object-oriented languages combine these by introducing the concepts of classes, objects and methods. In current networks packets are passive entities carrying data. From a router's point of view the payload has no semantics, it is just a sequence of bits. An interpretation of the data is only performed by the applications at the end nodes. This prevents a network from performing content- related actions, such as dropping B-frames, but not I-frames of an MPEG video in the case of network congestion. We claim that network performance can be improved in many ways if the semantics of the data is made available to the internal nodes. The lack of knowledge prevents intermediate nodes from performing *context-specific* actions on the packets. This is a major obstacle for introducing more intelligence into the network.

The main idea of active networks is to enable the network to perform *context-specific computations* on the data in the packets. Thus packets are transformed into objects including operations to be performed on them. In doing so, packets are converted from passive chunks of data to objects with specific semantics. The nodes of the network now need to have the means to execute the semantic actions. This requires computational power beyond that offered by current nodes.

But active networks also have inherent drawbacks. Since a node loads and executes foreign code at run-time there is a serious security exposure (the Trojan Horse problem). Also, since the run-time code must be portable, it will typically be less efficient than code written and compiled specifically for the hardware of a node. And the resource management problem within the nodes becomes much harder: It is possible that the code loaded for one application stream competes with the code for another simultaneous application stream for buffer space, CPU cycles, etc. We will discuss these issues in detail when we present the ANTS and M0 active network architectures.

The main goal of our work is to gain practical experience with two major active network systems, the ANTS system developed at MIT [WGT98] and the M0 system developed in Switzerland [Tsch97]. We have installed both systems on a network of Sun workstations, and we have implemented two experimental multicast protocols on each system. Both protocols inherently require processing within the internal nodes of a network and are thus good examples for our purpose. The first protocol is a robust audio protocol, adding a link-specific degree of redundancy to an audio packet stream, depending on the observed transmission quality. The second protocol transmits a layered, rate-adaptive multicast video stream. Here the idea is to optimize link load in the multicast tree by only forwarding as much of the video bit rate as at least one of the downstream receivers needs.

The remainder of this paper is structured as follows. In Section 2 we present the ANTS and M0 architectures in detail. Section 3 introduces the two protocol examples and their implementation. In Section 4 we discuss architectural insights, compare the performance of the two protocols on both systems, and report the lessons we learned. Section 5 presents related work, and Section 6 concludes the paper.

2 ANTS and M0: Two Architectures for Active Networks

ANTS (Active Network Toolkit System) is a toolkit for prototyping active network applications. It was developed by the TNS group at MIT [WGT98]. The description of the architecture and the code used in our project are based on the release of September 1997. ANTS is a distributed system running in user space on top of UDP. It is programmed in Java.

M0 (M-zero) was designed and implemented by Ch. Tschudin [Tsch93][Tsch97]. Its purpose is to provide a testbed for mobile code, with applications in networking and distributed systems. The M0 prototype runs on workstations over UDP, Ethernet or serial lines. It is programmed in the M0 language, which has a flavor similar to Postscript.

In principle active network architectures can be classified into those where each packet carries its own code, to be executed as it passes an active node, and those where code is cached in a node and only loaded on demand. M0 falls into the first class, ANTS into the second.

In the layered network model the active network layer replaces the IP layer, i.e. handles the processing and forwarding of datagrams at layer 3. But since layer 3 is usually in the OS kernel, and kernel programming is quite tricky, the active network prototypes run in user space on top of communication sockets.

2.1 The ANTS Architecture

The main purpose of ANTS is to enable an easy development and deployment of network protocols. The nodes in ANTS are called *active nodes*. Instead of passive packets ANTS has *capsules* which trigger specific processing when passing an active node. The piece of code to be executed is identified by a reference to the forwarding routine in the header field.

In ANTS code is loaded on demand by a sequence of capsules called a *code group*, a collection of related capsule types whose forwarding routines are transferred as a unit by the code distribution system. A *protocol* is a collection of related code groups that are treated as a single unit of protection by the active nodes.

2.1.1 Node Structure

An active node in ANTS has two caches, a code cache storing Java byte code, and a ‘node cache’ storing data. In addition it has a classical routing table that indicates the next hop to be taken to reach a destination node. When a capsule arrives at a node the channel thread picks it up and processes it until completion. Capsules have the right to spawn their own threads. The *evaluate* method of the class of which the capsule object is an instance is executed. Usually it performs some processing on the capsule’s content and forwards it to another node or delivers it to an application. New capsules of the same protocol can also be generated and injected into the network. The structure of an active node is shown in Figure 1. For a more detailed description the reader is referred to [WGT98].

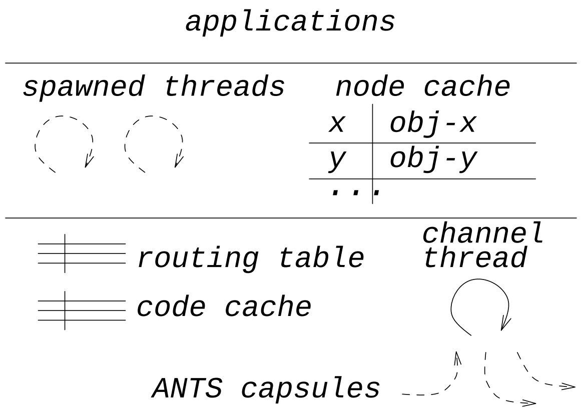

Figure 1: The structure of an ANTS node

All protocol code is written in Java using the ANTS API. An instance of the class *node* represents the local runtime for an active node. This class offers services that can be used during the processing of a capsule: access to the routing table, a cache for soft-states and registration of ANTS protocols.

2.1.2 Packet Structure

As mentioned above, packets are called *capsules* in ANTS. The structure of a capsule is very simple: It carries an identifier for its protocol and particular capsule type within that protocol, source and destination address, the remaining resource credits, the address of the previous active node, and application-specific data. Hence there is little overhead.

The ANTS Java API provides the abstract class *Capsule* for this representation, and user-defined capsule types must be subclasses of this class. The semantics of a capsule is determined by the method *evaluate*. It is called upon arrival of the capsule at an active node. Thus an application class must provide an implementation of this method.

In ANTS serialization and deserialization methods must be implemented, i.e., a bit stream representation of the capsule's data structure must be defined by the programmer.

2.1.3 Dynamic Code Management

The code representation in ANTS is the Java byte code format. If the code required by a capsule is found in the code cache, it is executed. If not, the active node generates a request capsule, sends it to the upstream neighbor and waits for the code group to be downloaded into the code cache. Once the code is there, it wakes up sleeping capsule threads, and they execute the code. The rationale behind this concept is that at least the originator of a capsule should have the code required for its processing. Thus new code is injected into the network by the application that created the capsule. The loading is performed with a specialized network classloader. Code is removed from the cache according to the LRU principle.

2.1.4 Resource Management

Controlling the resources of an active node is the basis for guaranteeing quality of service. In ANTS each capsule carries a Time-To-Live (TTL) field initialized at creation time. The value is decreased every time a node puts data into the cache, generates a new capsule, or upon transfer to another node. Capsules with a negative TTL value are discarded. A capsule cannot access its own TTL field; this is an example where security is based on an implicit feature of the programming language. If a capsule spawns a child capsule the remaining TTL is distributed over the two. There is no constraint on the size of the data put into the cache by a capsule. Furthermore, there is no restriction on the processing time for each capsule.

2.1.5 Security

Security is a very critical issue in active networks since foreign and unknown code is executed in the nodes. One of the foundations for the security in ANTS lies in the Java system itself. Using a high-level programming language with well defined access rules has many advantages:

- Capsules can only be manipulated through the public interface provided.
- The services an active node offers are also clearly defined and cannot be changed by a capsule.
- Essential methods can be declared final such that subclasses cannot re-implement them.
- The Java virtual machine performs byte code verification to check whether the code comes from a compiler conforming to the language specification.
- The concept of the security manager of Java can be used to tailor the access of the capsules to the services of a node.

But active networks introduce other security risks which cannot be handled in a such a straightforward manner. An example is protocol spoofing. To prevent this the ANTS system implements a clever security check: each capsule carries an identifier of its protocol and particular capsule type. The identifier is based on a fingerprint of the protocol code: it is computed as a hash function over the code itself. Thus the probability of a capsule invoking the wrong piece of code is negligible. Some aspects such as name conflicts still have to be solved in the ANTS system.

2.2 The M0 Architecture

What is a capsule in the ANTS architectures is called a *messenger* in the M0 system: Messengers are programs exchanged between M0 nodes. Messengers were proposed as a replacement of the classical message exchange paradigm used in networks today; they favor an entirely instruction-based way of communication. The M0 environment is an implementation of this approach. It continues to serve as an exploration tool for finding the minimal services an active node should provide, and as a programming environment where instruction-based communications can be studied for active networking, distributed operating systems and distributed artificial intelligence.

2.2.1 Node Structure

Basically there are only four major elements inside an M0 node: concurrent messenger threads, a shared memory area, a simple synchronization mechanism (thread queues), and channels towards neighboring M0 nodes (see Figure 2).

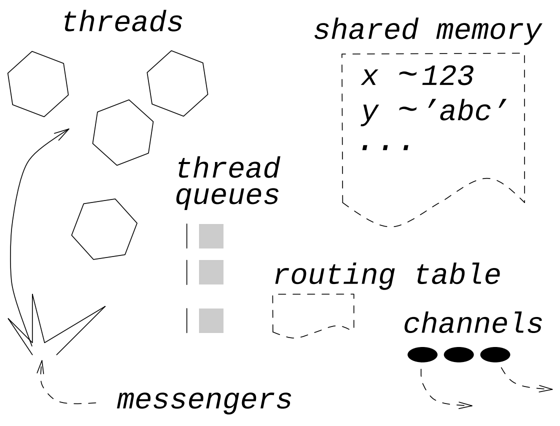

Figure 2: The structure of an M0 node

On arrival, each messenger is executed by an independent and anonymous thread of control. These threads have their own private memory space and are fully protected from each other - they have no identifier under which they could be addressed. Messenger threads can coordinate their activities through the shared memory area where they can deposit arbitrary data structures under self-chosen names so that other threads can access them. Thread queues are a way to serialize the execution of threads in order to avoid race conditions but can also be used for more general signaling purposes. Channels enable messenger threads to send new messenger packets to neighboring nodes: in the basic model routing functionality is not provided, so M0 nodes only talk to neighbors. The current M0 implementation maps messenger transmission to UDP, Ethernet or serial line communications. For the multicast applications we describe in this paper we have added support for an ANTS-like network with routing tables at the messenger level.

2.2.2 Packet Structure

M0 packets have a very simple format: a header, a code payload and an optional data payload. The header contains a version field, length information and a checksum that covers the header and the code but not the data. Packets with an invalid checksum are simply dropped which is consistent with the unreliable datagram semantics of the M0 channels.

A messenger's code field contains the program that the M0 platform will execute. Messenger code is written in a PostScript-like language, the M0 language. It is a high-level language that inherits from PostScript the main concepts of operand, dictionary and execution stack as well as the main data manipulation and flow control operators. It lacks everything related to rendering fonts and images. The M0 language also departs from PostScript with respect to the messenger- specific operators and a few new data types as well as the syntax. Most standard operators have single-letter names (which the programmer can easily redefine) that can be put back to back, yielding compact programs (very similar to bytecode). The M0 interpreter itself is written in C.

2.2.3 Dynamic Code Management

M0 deliberately has no explicit code caching or code loading functionality. The basic execution model simply assumes that code is shipped with *every* messenger. This works quite well for small protocols where the code is only a few hundred bytes long. For more important code sizes messengers implement their own caching method by storing the code in the shared memory area of a node under a chosen name (usually some random key, but this could also be the code's MD5 fingerprint): subsequent messengers just carry this reference inside a minimal instruction sequence that looks up the stored code and executes it. Note that a code deployment scheme can be implemented at the messenger level. Such schemes can also be shared with other messenger-based applications. This, however, requires some agreements about the way the code pieces are internally organized, the way it should be distributed (prefetching or on-demand, best-effort or reliable), and the policy how long it should remain in the store (i.e., who will pay for its storage). The point of view of the M0 designers is that it is impossible to devise a dynamic code management scheme that suits all needs, so one better exploits the full flexibility of mobile code.

2.2.4 Resource Management

Each M0 node manages its internal resources independent of other nodes. M0 relies on an economy-based model of resource allocation: all resources have price tags which depend on the node's actual load for a given resource, but also on the demand and offer from the running threads. Messenger threads are charged for their activities. When they run out of money they are silently removed from the system. On arrival, each messenger thread obtains an account with some start money. The amount is sufficient to do some exploration inside the node and eventually send out another messenger.

Accounts are also used for controlling the number of entries inside the shared memory area. Each entry must be "sponsored" by an account: by default this is the account of the thread creating the entry, but messengers can also add sponsoring accounts to entries they did not create. Periodically, the system charges the sponsoring accounts depending on the amount of shared data space they sponsor. If for an entry there is insufficient money left on its sponsoring accounts, the entry is removed. This sponsoring model implements in fact a user controllable ‘memory decay’ or ‘soft state’ mechanism.

2.2.5 Security

In M0, emphasis is put on building security *with* messengers instead of providing rich services at the system level. There is no authentication between M0 nodes, nor has a messenger some identity attached to it that would allow authentication. Safety-related questions on resource consumption have to be handled by controlling the flow of money. Messengers can effectively protect themselves against other messengers by having full control on which information they pass on to others in which way. Messengers cannot be killed by others simply because there are no thread handles or identifiers - the only way that the anonymous threads can interact is via the shared memory area and the synchronization mechanisms where in both cases a thread can control the degree of involvement and risk it is willing to accept. Access control for node-specific system resources is controlled by some agreement between a messenger and the system (e.g., a password). What M0 does provide are some basic cryptographic operators that can be invoked by a messenger. Currently these are DES and the MD5 hash function.

2.3 Summary of Features of ANTS and M0

We summarize the most important features of ANTS and M0 in Table 1.

Table 1: Summary of the main features of ANTS and M0

	ANTS	M0
Runtime Environment	Java Virtual Machine	M0 interpreter
Installation Require-ments	JDK 1.0 or higher	BSD or SVR4 UNIX, ANSI-C
Programming Language	Java	M0
Link Layer	UDP	UDP, Ethernet, serial lines
Code Distribution	system-supported, separated from normal capsules, code is cached	each messenger carries it code, cache can be implemented by messenger
Lifetime of capsule/msgr	user-defined TTL	potentially unlimited
Procreation limits	decrementing TTL for creating new capsule	none. new start money on arrival, money can be pooled
Cache usage	decrementing TTL for entering data	load-dependent prices
Cache removal policy	user-specified TTL, LRU replacement policy	sponsoring of entries determines lifetime
CPU cycles	no control	limited by available money
Protection against forbidden actions	based on Java security mechanisms	based on M0 interpreter
Protection against code spoofing	via fingerprint, hashed over the code	messengers carry their own code; no further system support

3 Two Protocol Examples

In this section we introduce two protocols for multimedia streams that we will use to demonstrate the usefulness of active networks. We have implemented both of them with ANTS and M0. The first protocol, Robust Multicast Audio, is an example of how the performance and efficiency of an existing protocol can be improved by adding application-specific compute power to internal nodes. The second, Layered Multicast Video, is an example of how active networks technology enables the quick development and deployment of a new protocol that optimizes network-internal bandwidth usage in multicast trees. Both applications involve continuous media, and both use the same multicast algorithms.

3.1 An Active Multicast Protocol

Our multicast tree management is based on the algorithm provided with [WGT98]. It uses two types of active packets (from now on we will use the term *active packet* as a common way to refer to capsules or messengers): *subscribe* active packets and *multicast* active packets. The *subscribe* active packets are sent periodically by the receivers towards the sender of the group they wish to join. Unlike [WGT98] we accumulate *subscribe* packets in intermediate nodes to avoid a *subscribe* implosion problem at the sender. These active packets install forwarding pointers in the nodes they traverse. These pointers are removed if they are not refreshed on time. The *multicast* active packets carry the real multicast data. They are routed along the distribution tree built by the *subscribe* active packets. The multicast implementation is thus based on the soft-state concept. Similar to [WGT98] we have not implemented an optimal tree routing algorithm; QoS-based multicast tree routing is beyond the scope of this paper. The paths of these active packets are shown in Figure 3.

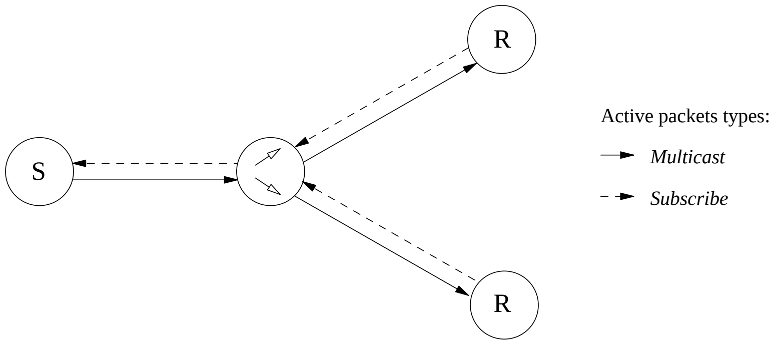

Figure 3: Multicast packet forwarding in an active node

3.2 Robust Multicast Audio

The first protocol we have implemented is Robust Multicast Audio (RMA). It is a protocol for improved-quality multicast audio transmission over best-effort networks, based on an encoder/decoder developed by M. Isenburg and H. Chordura at ICSI [Ise97]. The encoding is based on wavelets, and the system is called WAR (Wavelet Audio Radio). In our protocol, the link between the audio server and each audio client is subdivided into several point-to-point links internal to the active network. On each internal link, the audio stream only carries the amount of redundancy optimal for the loss currently observed on that link. On an incoming link the active node reconstructs the original data, on the outgoing links it adds the appropriate amount of redundancy. Figure 4 illustrates the link-dependent redundancy in the RMA protocol.

Compared to the classical end-to-end solution one of the advantages of this protocol is that it provides better performance since the losses on each internal link are recovered independently and thus do not add up. Another advantage is that redundancy is only added on those internal links where it is actually required, leading to a more efficient global use of the network resources than end-to-end redundancy.

The implementation of the RMA protocol uses three types of active packets: the *audio* active packets that carry the audio data, the *redundancy request* active packets that inform the active nodes about the losses on the internal point-to-point links, and the *audio subscribe* active packets that are used for multicast group subscription. The *audio* active packets are grouped in sequences, each consisting of $N + R_i$ active packets, where N is the number of original audio packets and R_i the number of redundancy packets added for internal link i. An active node waits until all the active packets belonging to a particular sequence have arrived. If losses occurred it reconstructs as many of the N original packets as possible. Before sending the packets on each of the outgoing links the node adds the appropriate amount of redundancy for that particular link. Each active node is instructed to monitor the losses in the incoming data stream and transmits this monitored value to the upstream node, using a redundancy request active packet. This packet will adjust the amount of redundancy added on that link in the future. Figure 4 shows the paths of these active packets and the algorithm executed at the active nodes.

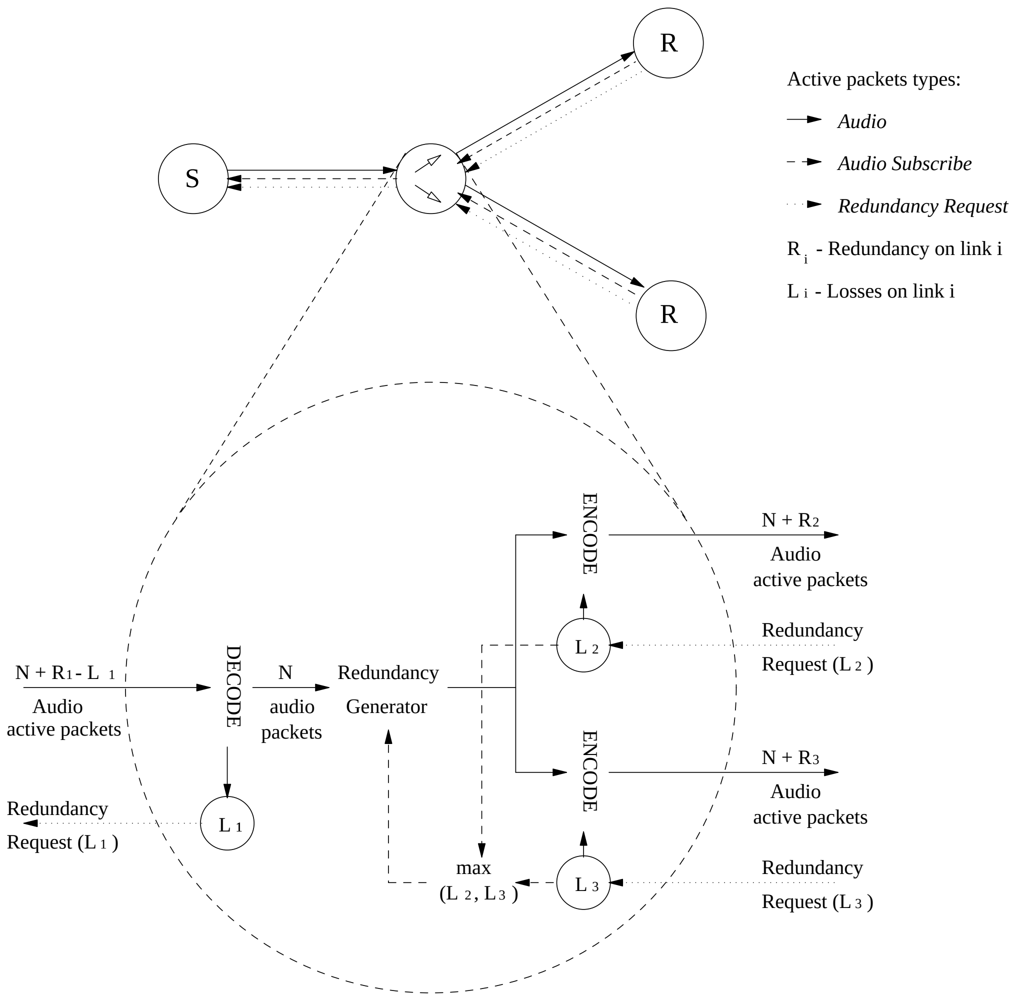

Figure 4: Link-dependent redundancy in the RMA protocol

In order to avoid modifying the original WAR application programs we integrated them into the active network through gateways. The client gateway is an application attached to an active node that provides a server interface for its communication with the WAR client. The server gateway plays the same role at the other end of the link, providing the interface to the WAR server. This is shown in Figure 5.

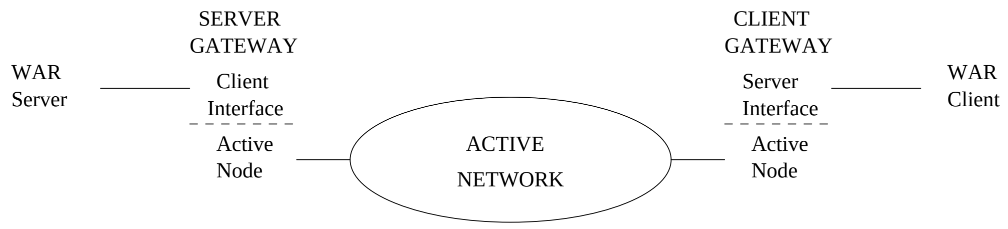

Figure 5: Gateways for the RMA protocol

The protection scheme used in our implementation is based on adding redundancy to a group of packets and waiting for all packets of a sequence at each active node for recovery, i.e. it is a classical forward error correction (FEC) scheme. The redundancy generated in our current version is based on a simple XOR scheme; a more sophisticated adaptive redundancy could also be easily implemented [LBET93]. A major advantage of radio broadcast is that delay is not a critical factor: a signal that arrives with a few seconds delay can still be said to be real-time since there is not immediate feedback from the receiver to the sender. This possibility of introducing delays allows to accumulate several seconds of encoded audio signal at the sender and make its transmission more robust by adding redundancy. In the original WAR system this redundancy is added at the end node; in our protocol it is added at the intermediate active nodes.

It would be very easy to replace the FEC scheme by an ARQ retransmission scheme and experiment with a variety of algorithms, measuring delays, throughputs etc. Since the nodes are active we can even *exchange algorithms* at runtime, a major advantage over passive networks. For example we could use ARQ when the link delay is very short, and FEC otherwise.

3.3 Layered Multicast Video

The second protocol we have implemented on top of the two active network systems is a layered, rate-adaptive multicast video protocol. One of the main problems of today’s MBone is that it cannot satisfy diverging requirements of a heterogeneous set of receivers because the multicast packet stream is transmitted to all receivers at the same rate and in the same format. In a typical multicast session some users might have high-speed end systems and high-speed access to the network while others might have low-end PCs and ISDN or modem connections. If the transmission rate of the source is high considerable bandwidth is wasted on links to low-speed receivers. If it is low high-speed end systems will experience low quality. It is desirable to set up a multicast tree with the optimal data rates for all receivers. This is illustrated in Figure 6.

Our layered multicast video system offers a solution to this problem. The video is encoded in multiple layers such that layer 0 provides a minimum quality stream and each layer i+1 adds more quality to layer i. Each active node participating in the multicast session understands the requirements of its subtrees and forwards only the corresponding video layers downstream. With active signaling packets the nodes inform their upstream neighbors: the layers they wish to receive are the maximum required by their subtrees and the local application (see Figure 6). The active nodes filter out layer packets at runtime according to this rule.

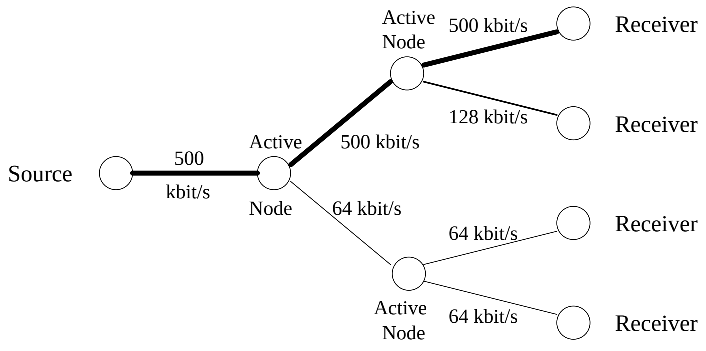

Figure 6: Distribution tree for layered multicast

The layered video encoder/decoder we have used to test this protocol is the Scalable Video Codec developed by W. Tan and A. Zakhor [TCZ96]. It is based on subband coding, a non-standard but very efficient video coding technology [TZ94]. Each packet in the encoded stream carries a layer identifier; the data in each packet belongs to one layer only. Thus the filtering in our active nodes is very simple: we can throw away entire packets if their data is not needed downstream. As a consequence our active filter code is very efficient.

The integration of this tool with the active network has been done in the same way as for the robust multicast audio protocol, i.e., using gateways for the interface between the tool and the active network. We use the same multicast tree mechanism: *subscribe* active packets to inform upstream neighbors about the layers needed, and *video multicast* active packets to carry the video data downstream. Bandwidth requirements are also implemented as soft-state, i.e. they time out and must be refreshed if still needed.

In the video application the flexibility of active networks would also allow us to add enhancements to the protocol. For example, in a lossy network environment we could add redundancy to protect the data, giving a higher priority to the most important pieces of data by adding more redundancy to the lower layer packets. This would be similar to ICSI's Priority Encoding Transmission (PET) approach [AL96] but adapted for layered video and implemented hop-by-hop rather than end-to-end.

We will now illustrate the code the programmer has to write for each of the two active network systems. As an example we present the routine that filters the *video multicast* active packets at run-time. Figure 7 shows the Java code running in an ANTS active node. As explained above this code piece is part of a protocol; it is downloaded on demand from the upstream neighbor and cached. Figure 8 shows the M0 code running in an M0 active node. This code is sent with each *subscribe* messenger, extracted and then executed in a local tread.

```
public boolean evaluate(Node n) {
  int layersRequested = 0;
  VideoMulticastInfo vmci =  n.getCache().get(group, target);
  if (vmci != null) {                    // Am I interested in this session?
    if (vmci.links != null)              // Do I have sons?
      for (int i = 0; i < vmci.links.length; i++) { // process each outg. link
        layersRequested = vmci.links[i].layers;
        if (layer < layersRequested)     // deliver only if interested
          n.routeForNode(this, vmci.links[i].address);
      }
         // if attached application is interested in layer: deliver
    if ((app = n.getApplication()) != null && app.getLayers() > layer)
      n.deliverToApp(this, dpt);
  }
  return true;
}
```

Figure 7: Code of a video multicast capsule in ANTS

```
# M0 code for the central multicast dispatch loop for layered video
# (using the long operator and variable names)
# variables:
#   leveldict           # dict with the downstream clients' requested layers
#   outdict             # dict with message queues to store new payloads
# on the operand stack (growing downwards):
#   the payload to be multicast
#   the payload's layer number
leveldict copy {        # loop over all clients
    2 index gt {              # if client request > layer number then:
      outdict 1 index get     #   get the client's message queue
      1 neg 4 index put       #   append the current payload to it
      0 setqueuestate         #   wakeup the client's downlink procedure
    }{                        # else:
      pop                     #   skip this client
    } ifelse
  } loop
```

Figure 8: Code of a video multicast messenger in M0

4 Experimental Results and Lessons Learned

The implementation of the two multicast protocols with the two active network systems was a very interesting experience. It provided us with concrete insights into the practical consequences of architectural decisions and enabled us to evaluate the performance.

4.1 Architectural Comparison

ANTS and M0 are similar in their basic approach to active networking. However they differ in several major architectural aspects such as programming language, application programmer interface and execution model which we will discuss in the following.

4.1.1 Language

The programming language used in ANTS is Java. Although capsules are processed in the same address space, an application programmer does not need not worry about uncontrolled manipulation of capsules: capsules in a node are objects that can only be manipulated by the public methods defined in the class. Developing an ANTS application is relatively easy because a user only has to write subclasses for given classes. Since development tools for Java are abundant, local testing and debugging is well supported. The Java skill base is increasing very quickly which allows protocol developers to concentrate on the protocol logic rather than new language concepts.

M0 on the other hand predates Java. PostScript had proven to be a successful portability technology. It is therefore quite logical to extend the approach of communicating with a printer or a screen to communication protocols in general. M0, like Java, is based on an interpreter which also supports multithreading. Compared to Java the code written in M0 is harder to understand, and the PostScript skill base is much smaller.

M0 also has the disadvantage of not being object-oriented; in recent years object orientation has proven to be a powerful software engineering paradigm. On the other hand the programmer has more flexibility in M0; for example dynamic code creation for compression or encryption purposes is easier.

4.1.2 Application Programmer Interface

A considerable advantage of ANTS is that applications can be written in Java and then execute in the same environment with the active network functionality: The application becomes an ANTS node. The M0 platform looks more like a router. The fact that the application code and the code for the active network nodes can be written in the same programming language avoids an `impedance mismatch' at the programming level; it also avoids time-consuming and error-prone data representation transformations. For the video multicast application, the M0 implementation used three different languages: M0 for the multicast protocol, C to implement the gateway between the existing video software and Tcl/Tk to add a graphical user interface.

4.1.3 Execution Model

Both systems follow the same model in that upon arrival of an active packet the corresponding code is executed. M0 creates an independent thread for each incoming active packet to perform this processing, thereby providing different address spaces. In ANTS there is (by default) a single thread called ChannelThread which is used by all capsules. This has a disadvantage: if the processing of a capsule takes a long time (or even worse, an infinite loop occurs) the node is blocked, and incoming capsules may be lost due to buffer overflow. It is up to the capsule programmer to spawn his/her own threads should the expected processing time be long. For our video application the processing load was very low, so no new thread was created. In the audio application a thread was not created for every capsule, but only for calculating the redundancy. This flexibility proved to be very useful in the ANTS applications. We observe a trade-off between better performance and the danger of blocking processes.

4.2 Packet Structure

ANTS and M0 rely on user-defined serialization, and in both cases it is easy to get things wrong. The ANTS API provides procedures to concatenate portable bit representation of simple base types. This proprietary approach should be replaced by the Java 1.1 serialization package. M0 provides a procedure for turning any simple data type into an M0 code string that is able to recreate the encoded value. Serializing a sequence of simple values typically consists of concatenating the code strings: after executing the full string all values will be found on the operand stack.

It is an open question whether fragmentation of large active packets should be under the control of the programmer. In our experiments we did not have the problem of limits in the size of an active packet: all ANTS capsules and M0 messengers fitted into a single UDP packet, which was the transport mechanism we used. Doing the same with Ethernet would not have been possible with ANTS because some code capsules would have been too large.

4.3 Dynamic Code Management

A major difference between ANTS and M0 is their respective approach to code distribution and caching. ANTS provides a code-on-demand mechanism and implements a *code follows the path of the capsule* policy. This has the advantage that the programmer does not have to program the code distribution for each new protocol. M0 has no system support for this: each messenger must contain the code it wants to execute. Messengers can implement their own code caching mechanism if desired.

M0's flexibility proved useful for the two multicast applications that we implemented. The upstream *subscribe* messengers that create the multicast tree are also responsible for code distribution; they install the code for the client-specific delivery of *multicast* messengers. The downstream *multicast* messengers consist of a very small lookup routine (12 bytes) for invoking this preinstalled code. In our opinion it is an open question whether code distribution by the sender or code distribution by the receiver is better for a receiver-oriented multicast scheme.

This is an example of how the code distribution mechanism provided by ANTS and M0 influenced our protocol implementations. Because M0 does not come with a standard code distribution protocol, no code-on-demand functionality is used: code is distributed in every *subscribe* which helps to keep the *multicast* messengers small. Because ANTS provides a convenient default code distribution mechanism, no attempts were made to implement code installation for downstream flows by upstream capsules. The difference in the code distribution philosophy is also visible at another level: because ANTS imposes that code executable by a capsule belongs to the same protocol it is not possible to have leaves push their proprietary delivery method into an already existing multicast tree. All possible `methods' have to be known at protocol registration time. In M0 this is not an issue, although some additional effort would be necessary for securing the interactions between multicast messengers and code installed by upstream packets.

4.4 Resource Management

In both architectures resource management is a subject for further study. In ANTS almost all critical resources are not covered yet. CPU time, for example, can not be bound, thread spawning is not monitored, bandwidth is not taken into account, nor is the amount of memory grabbed by a capsule. The handling of resource credits is simple; operations like storing data in the node cache always cost one unit, regardless of the current memory and cache utilization. In M0 the resources CPU time, bandwidth, thread creation and memory usage (local to a thread as well as shared memory) are all monitored. Resource usage costs a price depending on the actual load for that resource.

A first difference is that ANTS uses a fixed-price model whereas M0 uses load-dependent pricing. The fixed-price model has the advantages of predictable cost and lower overhead whereas the variable-price model has the advantages of signaling the load and money-based priorities. In both models it remains unclear how the initial amounts of money are assigned to the applications creating the active packets.

4.5 Security

The main problems for a secure execution environment in an active node are:

1. to provide a separate execution space for each active packet,
2. to control the allowable actions performed during packet processing,
3. to load the code securely over a public network,
4. to identify the code to be used for an incoming active packet, and
5. to identify the producer of an active packet.

For problem 1 both systems provide separate execution spaces for each active packet. The only security exposure is the cache or shared memory in a node. Packets would have to guess the keys generated by other packets in order to intrude.

To solve problem 2 ANTS provides its own security manager based on the Java security model to restrict allowable operations. It is not necessary to restrict operations explicitly in M0 because the language only provides allowable operations (no file I/O, etc.).

There are no concepts to solve problem 3 in ANTS; in its current version it ships the code over the network in plain format. In M0 it is up to the programmer to encrypt the code.

For problem 4 ANTS uses fingerprints computed over the byte code to prevent code spoofing. In M0 there is no such protection for messengers.

To deal with problem 5 each ANTS capsule has a source address that cannot be changed from within the ANTS system. In M0 there is deliberately no source address field in messengers.

4.6 Runtime Efficiency

We evaluated the performance of our protocols in terms of throughput and overhead. Because ANTS and M0 are active network nodes running at the application level, their performance cannot be compared with the performance of an active or passive network running at the network layer. The purpose of our performance tests was to gain insight into the impact of the processing required by our protocols. We compared the relative performance of the two active network systems, and our measurements gave us hints how an active node reacts to changing processing requirements.

We ran most of our experiments on a single 10 Mbps Ethernet segment using Sun SPARC5 workstations under Solaris 2.5. In those cases where the Ethernet became the bottleneck we used a 155 Mbps ATM network. The Java version was JDK 1.1 without a JIT compiler. The experimental setup consisted of a workstation generating packets, an intermediate workstation routing these packets, and a third workstation receiving them and making the measurements. We wanted to measure the maximum throughput the intermediate active node could route. In this scenario we tested both of our multicast protocols: we measured the maximum throughput that an active node could route with less than 0.1% losses. The results are shown in Table 2. The generation of packets at the source was done with bursts of packets with intervals of 20 ms between them. In the audio protocol we enforced the active node to always add 50% XOR redundancy in order to evaluate the impact of the redundancy computation (the throughput given in Table 2 includes these redundancy packets).

Table 2: Measurements for video/audio active packets

Throughput [pkts/s]	**ANTS**	**M0**
Video (1324 bytes/pkt)	133	195
Audio (1090 bytes/pkt)	100	155

In order to better understand the decrease in throughput due to active network technology we also implemented small application programs that forwarded UDP packets with the same payload through the network. We did that in C and in Java. We also wrote versions of a protocol for ANTS and M0 that only forwarded packets without any further processing. The measurement results are shown in Table 3. We see that M0 provides a better throughput than ANTS. This is mainly due to the programming language used to implement the interpreter: C in the case of M0, Java in the case of ANTS. However M0 pays a higher price for performing computations on the active packets in the node. This can be observed from the fact that its throughput decreases considerably in the full video and audio protocols. It should be noted that the M0 node ran with resource management enabled while the ANTS node had virtually none.

Table 3: Measurements for packet forwarding

Throughput [pkts/s]	**C ATM**	**C Ethernet**	**Java Ethernet**	**ANTS Ethernet**	**M0 Ethernet**
Payload 1324 bytes	1000	400	200	166	360
Payload 1090 bytes	1100	500	200	166	360

Active networking introduces two types of overhead: code shipping and additional header fields. For our protocols the size of the code to be shipped was 800 bytes in M0 and between 1600 and 4000 bytes in ANTS. However, since the code is shipped infrequently (in ANTS it is loaded at the beginning of the session, in M0 it is sent with every *subscribe* capsule), this is not much of a burden to the network.

The overhead carried with each active packet was 68 bytes in ANTS (all of them in the header) and 32 bytes in M0 (20 bytes in the header and 12 bytes of code). Considering that the data payload is 1090 bytes in the audio and 1324 bytes in the video this overhead is minimal.

4.7 Code Size and Reusability

The size of the code we had to write was relatively small. Table 4 gives an overview for the ANTS and M0 implementations.

Viewing the network as a distributed programming system introduces the software engineering aspect into protocol development. In designing our two applications we found that they have a lot in common, therefore code could be shared. Since both protocols are multicast protocols, the multicasting algorithm could be separated and the corresponding code reused.

Table 4: Code sizes (in lines of code)

	ANTS			M0		
	Capsule code (total) in Java	**Gateway code (client and server) in Java**	**User Interface in Java**	**Messenger code (total) in M0**	**Gateway code (client and server) in C**	**User Interface in Tcl/Tk**
Audio Multicast	500	280	450	170	460	---
Video Multicast	260	150	280	170	600	70

The use of Java in ANTS made reuse of the multicast code very easy: A general multicast capsule class was implemented and refined for both applications. This would even allow us to experiment with different multicast protocols by just plugging in the new classes. It would also be possible to reuse a multicast class developed by a third party in our applications, as long as this class adhered to a fixed interface. We expect to see special APIs for protocol development in active networks in the future. These will make it even easier to develop and exchange new applications.

5 Related Work

The term *active network* is relatively young, but quite a few groups are already working on the topic. Good overviews of active network projects can be found in [Ten97] and [Mun97].

There are many similarities between active networks and *mobile agent systems (mobile code).* A good overview of mobile agent projects can be found in [RP97]. The main difference is that active networks use the concepts for network layer processing whereas mobile agent systems run as application programs. Both have dynamic code deployment, caching, resource control and many other components on common.

In the area of multimedia communications the idea of media scaling and media filtering has found some attention in recent years although the algorithms and protocols proposed for it have never made their way into network products. For example a media scaling mechanism for MPEG video could be the dynamic adjustment of the quantization parameter at the encoding site as a function of the QoS parameters of a link. An example for media scaling can be found in [KW95], and filtering mechanisms are described in [YMH96].

The RSVP protocol is designed to carry resource reservation packets in the Internet. The protocol itself does not specify what resources it deals with. The designers explicitly mention the inclusion of packet filters into the data path of an IP stream at each network node; all packets passing through a particular filter share a particular resource [ZDE93, BZE96]. In principle RSVP could be used to pipe an incoming packet stream through a video layer filter very similar to ours. However, the difference to our system would be that with RSVP, the code for the filters has to be installed in all routers *beforehand*. RSVP can only turn it on or off for a particular IP packet stream; there is no way to dynamically load filter code. Thus the overall design is much less flexible than ours. In a recent paper R. Wittmann and M. Zitterbart propose an active network extension to RSVP [WZ97].

In his dissertation S. McCanne of UC Berkeley explicitly addresses the composition and transmission of a layered video stream for multicast networks [McC96]. McCanne has developed his own layered video encoding scheme, based on a combination of DCT and wavelets (unlike the subband coding used in our experiments, see [TCZ96][TZ94]). His system was designed for operation over multicast IP, without special code in the routers. His key idea is to transmit the different layers of video to different multicast groups; the more groups a user joins, the better his video quality will be. An advantage of this approach is that the internal network nodes can be normal multicast IP routers, forwarding the packets according to their group addresses. But the fact that the senders and receivers have to use multiple IP groups for a single logical video stream is not very clean conceptually; a group should be a set of nodes interested in a particular content and not a set of nodes receiving a particular layer of a video stream. For example, as a consequence of the multi-group encoding, the session directory sdr will contain several entries for the same transmission, one for each layer of video, and the receivers have to deal with that. Another problem is that too little semantics are known to the internal nodes. For example, with the subband coding of [TCZ96], if the packets for layer n are lost on a link, all packets of higher layers can be discarded downstream as useless, but the multi-group scheme is unable to handle such an optimization.

Some researchers oppose to the use of active network technology in the main data path; they claim that dynamically loadable code can never be efficient enough. In [Bra97] B. Braden describes a *signaling* protocol based on active network technology; for signaling the code efficiency is not as critical as for the data path. Using active packets for network diagnostics, monitoring and auto-configuration is proposed in [JP97].

6 Conclusions

We have successfully implemented a robust audio multicast protocol and a layered video multicast protocol with the two active network systems ANTS and M0. It was surprisingly easy to get the code to work; all four implementations were done within four weeks by a team of three people. Runtime efficiency was much better than expected, but more detailed experimental results are still needed.

Our main conclusion from the comparison between ANTS and M0 is that there is a tradeoff between performance and other aspects such as security and resource management. The M0 architecture pays more attention than ANTS to the latter aspects but also receives a higher penalty for processing at the nodes. Another issue is the programming language: ANTS obtains portability and security facilities from Java but also pays a higher price in performance due to Java. We observed that performance was sufficient to carry multimedia data streams.

From our performance measurements we conclude that active network technology can be used not only for signaling protocols or network management but also in the data path of novel application protocols as long as the operations to be performed on each active packet remain simple.

With the Java language in the ANTS system, the network is programmed in the same way as the application; this will allow to implement future network applications in an integrated manner, with part of the code running in internal nodes and other parts in the end systems. We consider this to be a powerful new programming paradigm for networked applications

The code deployment mechanism in M0 is more flexible than the ‘code follows the path of the capsule’ mechanism of ANTS. For the implementation of receiver-specific filters in the multicast tree this proved to be an advantage.

We observe that multicast protocols are good candidates for active network technology. Compared to point-to-point protocols they inherently require more processing in internal nodes. For example it is easy to implement reliable multicast protocols with active networks.

We do not expect active network technology to completely replace existing high-performance implementations of protocols such as classical IP or multicast IP. But we can imagine a router architecture where classical packet streams are handled by passive code as usual; for application-specific protocol processing or experimental protocols the router would have a secure ‘sandbox’ for loadable code.

Acknowledgments
Martin Isenburg and Hartmut Chodura gave us their WAR code for robust audio encoding/decoding which we gratefully acknowledge. We would also like to thank Avideh Zakhor and Wai Tian Tan of UC Berkeley for letting us use their layering video encoder/decoder. Without their spontaneous help the experiments described in Section 3 would not have been possible.

References

[AL96] A. Albanese, M. Luby: PET - Priority Encoding Transmission, in: High-Speed Networking for Multimedia Applications, Kluwer Academic Publishers, Boston, March 1996.

[Bra97] B. Braden: Active Signaling Protocols. URL: ftp://ftp.isi.edu/rsvp/active_signaling/

[BZE96] B. Braden, L. Zhang, D. Estrin, S. Herzog, S. Jamin: Resource Reservation Protocol (RSVP), Version 1 Functional Specification, Internet Draft, IETF, August 1996.

[Ise97] H. Chodura, M. Isenburg: WAR: Wavelet Audio Radio, URL: http://www.icsi.berkeley.edu/~isenburg/paper.ps.gz

[JP97] A.W. Jackson, C. Partridge: Smart Packets – A DARPA-Funded Research Project, March 1997, URL: http://www.net-tech.bbn.com/smtpkts/baltimore/index.htm

[KW95] T. Käppner, L. Wolf: Media Scaling in Distributed Multimedia Object Services, in: Proc. Multimedia Advanced Teleservices and High-Speed Communication Architectures, R. Steinmetz (ed.), Springer LNCS 868, 1995, pp. 34-43

[LBET93] B. Lamparter, O. Böhrer, W. Effelsberg, V. Turau: Adaptable Forward Error Correction for Multimedia Data Streams, Dept. of Computer Science, University of Mannheim, TR 93-009, December 1993, URL:http://www.informatik.uni-mannheim.de/informatik/publications/index.publications.html

[McC96] S. R. McCanne: Scalable Compression and Transmission of Internet Multicast Video. PhD dissertation, report No. UCB/CSD-96-928, Computer Science Division, UC Berkeley, 1996, URL: http://http.cs.berkeley.edu/~mccanne/phd-work/

[Mun97] S. Munir: Active Networks - A Survey. URL: http://www.cis.ohio-state.edu/~jain/cis788-97/active_nets/index.htm

[RP97] K. Rothermel, R. Popescu-Zeletin (Eds.): Mobile Agents. Proc. 1st International Workshop on Mobile Agents, Berlin, April 1997, Springer LNCS 1219

[TCZ96] W. Tan, E. Chang, A. Zakhor: Realtime Software Implementation of Scalable Video Codec, in: Proc. Int. Conf. on Image Processing, Lausanne, Switzerland, 1996. Also URL: http://www-video.eecs.berkeley.edu/~tan/icip96.ps.gz

[Ten97] D. Tennenhouse et al.: A Survey of Active Networks Research, IEEE Communications Magazine, Vol. 35, No. 1, January 1997, pp. 80-86

[TMM96] Ch. Tschudin, G. Di Marzo, M. Muhugusa and J. Harms: Welche Sicherheit für mobilen Code? In: Proc. Fachtagung Sicherheit in Informationssystemen (SIS'96), Wien, March 1996, vdf-Verlag, pp. 291-307 (in German)

[Tsch93] Ch. Tschudin: On the Structuring of Computer Communications. PhD Thesis No. 2632, Université de Genève, 1993, URL: ftp://cui.unige.ch/pub/tschudin

[Tsch97] Ch. Tschudin: The Messenger Environment M0 - A Condensed Description. In: Mobile Object Systems - Towards the Programmable Internet, J. Vitek and Chr. Tschudin (eds), Springer LNCS 1222, 1997, pp. 149-156

[TZ94] D. Taubman, A. Zakhor: Multirate 3-D Subband Coding of Video, in: IEEE Trans. Image Processing, Vol. 3, No. 5, 1994, pp. 572-588

]WGT98] D.J. Wetherall, J.V. Guttag and D.L. Tennenhouse: ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. Submitted to IEEE OpenArch 98, San Francisco, April 1998. Also URL: http://www.tns.lcs.mit.edu/publications/openarch98.html

[WZ97] R. Wittmann, M. Zitterbart: AMnet: Active Multicast Network, in: Proc. 4th COST237 Workshop 'From Multimedia Services to Network Services', Lisboa, December 1997, Springer LNCS, to appear

[YMH96] N. Yeadon, A. Mauthe, D. Hutchison, F. Garcia: QoS Filters: Addressing the Heterogeneity Gap, in: Proc. Interactive Distributed Multimedia Systems, B. Butscher, E. Möller, H. Pusch (eds.), Springer LNCS 1045, Berlin, 1996, pp. 227-244

[ZDE93] L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala: RSVP: A New ReSerVation Protocol, in: IEEE Network, September 1993