

Multicasting Multimedia Streams with Active Networks*

Albert Banchs
Telefonica I+D
Emilio Vargas, 6
E – 28043 Madrid, Spain
banchs@tid.es

Christian Tschudin
University of Zurich
Computer Science Department
Winterthurerstrasse 190
CH – 8057 Zürich, Switzerland
tschudin@ifi.unizh.ch

Wolfgang Effelsberg
University of Mannheim
Praktische Informatik IV
D – 68131 Mannheim, Germany
effelsberg@pi4.informatik.uni-mannheim.de

Volker Turau
FH Wiesbaden
Fachbereich Informatik
Kurt-Schumacher-Ring 18
D – 65197 Wiesbaden, Germany
turau@informatik.fh-wiesbaden.de

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 Node 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 networks, ANTS, M0, robust audio, scalable video, layered video.

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 through 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.

The next generation of computer networks will have to handle a large 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 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 services in internal network nodes (stream-specific processing, network management) seems very desirable.

Packets in current networks 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. The lack of knowledge 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. One central idea of active networks is to transform packets into objects that include the operations to be performed on them. In doing so, packets are converted from passive chunks of data to objects with specific semantics. We claim that network performance can be improved in many ways if the semantics of the data is made available to the network's internal nodes.

But active networks also have inherent drawbacks. Since a node loads and executes foreign code at runtime 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

*This research was done when the authors were at the *International Computer Science Institute*, 1947 Center Street, Berkeley, CA 94704, USA, <http://www.icsi.berkeley.edu>

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 [21] and the M0 system developed in Switzerland [18]. 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. Receivers announce their requirements to their upstream node. Each active node requests the maximum of the requested rates to its upstream node and only forwards as much of the video bit rate as a subtree 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 Node Transfer System) is a toolkit for prototyping active network applications. It was developed by the TNS group at MIT [21]. 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 C. Tschudin [18]. Its purpose is to provide a testbed for mobile code, with applications in networking and distributed systems. The M0 prototype also runs in user space. 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.

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. All protocol code is written in Java using the ANTS API.

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.

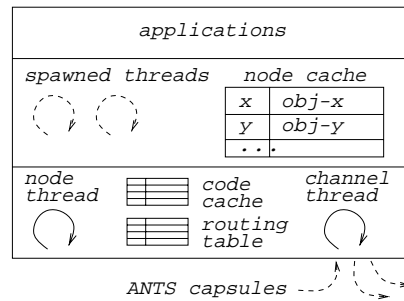


Figure 1. The structure of an ANTS node.

An instance of the Node class 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

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. User-defined capsule types must be subclasses of this class and implement the “evaluate” method as well as serialization and deserialization methods.

That is, the 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 that 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 of an active node 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 straight-forward 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 that is based on a cryptographic fingerprint of the protocol code. Thus the probability of a capsule invoking the wrong piece of code is negli-

gible. 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: it is a program exchanged between M0 nodes. Messenger exchange was proposed as a replacement of the classical message exchange paradigm used in networks today; it favors an entirely instruction-based way of communication [17]. The M0 execution environment is an implementation of this approach.

2.2.1 Node Structure

There are 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).

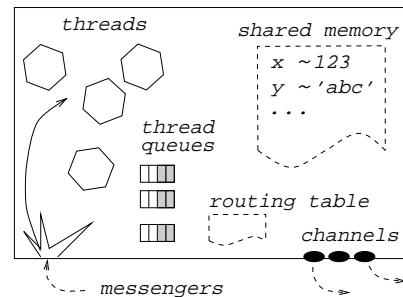


Figure 2. The structure of an M0 node.

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. Channels enable messenger threads to send new messenger packets to neighboring nodes. The current M0 implementation maps messenger transmission to UDP, ANON [19], Ethernet or serial line communications. For the multicast applications described 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 field and an optional data payload. The code field contains the program that the M0 platform has to execute. Messenger code is written in the compact 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, but 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. 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: it is argued that it is impossible to devise a dynamic code management scheme that suits all needs (prefetching or on-demand, best-effort or reliable, policy how long code should remain in the cache etc.). The basic execution model simply assumes that code is shipped with every messenger. This works quite well for small protocols where the compact code is only a few hundred bytes long. For more important code sizes the programmers have to implement their own caching method by storing the code in the shared memory area of a node under a self-chosen name (usually some random key): subsequent messengers carry this reference inside a minimal instruction sequence that looks up the stored code and executes it.

2.2.4 Resource Management

Each M0 node manages its internal resources independently of other nodes. M0 relies on an economy-based model of resource allocation [16]: 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. 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 a user controllable 'memory decay' mechanism because messengers can add and/or refill sponsoring accounts.

2.2.5 Security

M0 puts emphasis on building security with messengers instead of providing rich services at the system level [20, 11]. 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

Table 1. Summary of main features of ANTS and M0.

	ANTS	M0
Runtime Environment	Java Virtual Machine	M0 interpreter
Requirements	JDK 1.0 or higher	UNIX, ANSI-C
Programming Language	Java	M0
Link Layer	UDP	UDP, ANON, ethernet, serial line
Code Distribution	system-supported, separated from normal capsules, code is cached	each messenger carries its 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
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
Host protection	based on Java security mechanisms	based on M0 interpreter
Preventing code spoofing	via fingerprint, hashed over the code	no system support

information they pass on to others in which way. M0 provides 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.

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 algorithm that we introduce in the following section.

3.1 An Active Multicast Protocol

Our multicast tree management is based on the algorithm provided with [21]. 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 [21] 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 softstate concept. The paths of these active packets are shown in Figure 3.

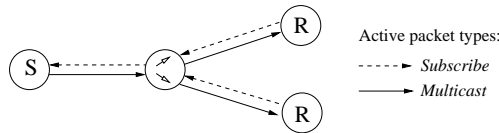


Figure 3. Multicast packet forwarding in an active node

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 [5]. The encoding is based on wavelets, and the system is called WAR (Wavelet Audio Radio). We used ANTS and M0 to actively multicast a Berkeley radio station’s music program.

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.

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 *subscribe* active packets that

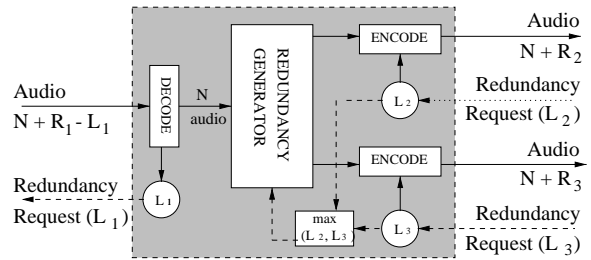


Figure 4. Link-dependent redundancy in the RMA protocol.

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 L 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 (see Figure 4).

In order to avoid modifying the original WAR application programs we integrated them into the active network through gateways. The *client side gateways* are applications attached to an active node that provide a server interface for their communication with the receivers’ WAR client software. The *server side gateway* is the root of the multicast tree: it plays the role of a WAR client in its interface with the WAR server. The main task of the gateways is to turn traditional packets into active packets and vice versa.

3.2.1 Discussion

Compared to the classical end-to-end solution one of the advantages of the RMA 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.

A major advantage of multicasting a *radio* program 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. In the original WAR system the 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. [7]. Since the nodes are active we can even switch 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

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. Considerable bandwidth is wasted on links to low-speed receivers if the transmission rate of an upstream multicast node does not match the limited downstream capacity. Therefore it is desirable to set up a multicast tree with the optimal data rates for all receivers. This is illustrated in Figure 5.

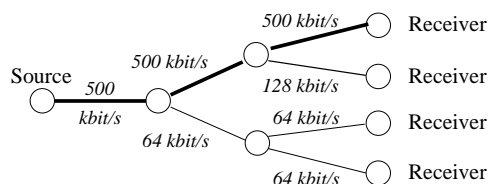


Figure 5. Distribution tree for layered multicast.

The second protocol we have implemented on top of the two active network systems is a layered, rate-adaptive multicast video protocol. 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. The nodes inform their upstream neighbors with active signaling packets about the layers they wish to receive (this is the maximum required by their subtrees and the local application). The active nodes filter out layer packets at runtime according to these requests.

We have used the Scalable Video Codec developed by W. Tan and A. Zakhor [13] in order to test this adaptive multicast protocol. The codec is based on subband coding, a non-standard but very efficient video coding technology [14]. 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.

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 [1] but adapted for layered video and implemented hop-by-hop rather than end-to-end.

In [2] we 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 runtime.

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 to 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.

4.2 Packet Structure

ANTS and M0 rely on user-defined serialization, and in both cases it is easy to get things wrong. M0 provides a procedure for turning simple data types 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. The ANTS API provides methods to serialize and deserialize simple types. Users must provide corresponding methods for their capsule classes. This can be difficult for complex classes. Using the Java 1.1 serialization package would ease this work and make applications more robust.

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.

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. Messengers can carry 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, it was possible to program the application such that the frequent multicast messengers become as small as possible. 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.

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.

ANTS' fixed-price model has the advantages of predictable cost and lower overhead whereas M0's variable-price model has the advantages of signaling the load and enabling money-based priorities. In both models it remains unclear how the initial amounts of money are assigned to the applications creating the active 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

4.5 Security

ANTS and M0 do not thoroughly handle security issues. We highlight some possible concerns.

Separate execution space for each active packet: 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.

Access control: 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.).

Secure code shipping: This concern is not addressed 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.

Code integrity: ANTS uses fingerprints computed over the byte code to prevent code spoofing. In M0 there is no such protection.

Code authentication: 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 first 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

Table 3. Measurements for packet forwarding

Throughput (pkts/s)	Payload 1324 bytes	Payload 1090 bytes
C (ATM)	1000	1100
C (ethernet)	400	500
Java (ethernet)	200	200
ANTS (ethernet)	166	166
M0 (ethernet)	360	360

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).

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 (Table 2). It should be noted that the M0 node ran with resource management enabled while the ANTS node had virtually none.

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 multicast 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. 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.

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 [15] and [9].

There are many similarities between active networks and mobile agent systems. A good overview of mobile agent projects can be found in [10]. 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 in 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 [6], and filtering mechanisms are described in [23].

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 [24, 4]. 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 [22].

In his dissertation S. McCanne of UC Berkeley explicitly addresses the composition and transmission of video stream for multicast networks [8]. He also uses a layered video encoding scheme [13, 14], but assigns these layers to individual IP multicast groups: the more groups a user

Table 4. Code sizes (in lines of code)

	Audio Multicast	Video Multicast
Capsule code (total) in Java	500	260
Gateway code (client and server) in Java	280	150
User interface in Java	450	280
Messenger code (total) in M0	170	170
Gateway code (client and server) in C	460	600
User interface in Tcl/Tk	n.a.	70

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. An inconvenience is that 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. A more severe limitation is that too little semantics are known to the internal nodes. For example, with the subband coding of [13], 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 [3] 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 [12].

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: performance was sufficient to carry multimedia data streams. From our 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 and experimental application protocols as long as the operations to be performed on each active packet remain simple. Multicast protocols are good candidates for active network technology. Compared to point-to-point protocols they inherently require more processing in internal nodes.

The comparison between ANTS and M0 revealed 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.

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 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 hybrid router architectures offering standard handling of packet streams as well as secure “sandboxes” for user provided code to process application-specific protocols.

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 Avidesh 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

- [1] A. Albanese and M. Luby. PET - priority encoding transmission. In *High-Speed Networking for Multimedia Applications*. Kluwer Academic Publishers, Mar. 1996.
- [2] A. Banchs, W. Effelsberg, C. Tschudin, and V. Turrau. Multicasting multimedia streams with active networks. Technical report, International Computer Science Institute, Berkeley, Nov. 1997. TR 97-050, <ftp://www.icsi.berkeley.edu/pub/techreports/1997/tr-97-050.ps.gz>.
- [3] B. Braden. Active signaling protocol. ftp://ftp.isi.edu/rsvp/active_signaling/ASP_overview.ps, June 1997.
- [4] B. Braden, L. Zhang, D. Estrin, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP), version 1 functional specification. Internet Draft, IETF, Aug. 1996.
- [5] H. Chodura and M. Isenburg. WAR: Wavelet audio radio. <http://www.icsi.berkeley.edu/~isenburg/papers/war.ps.gz>, Aug. 1997.
- [6] T. Käppner and L. Wolf. Media scaling in distributed multimedia object services. In R. Steinmetz, editor, *Multimedia Advanced Teleservices and High-Speed Communication Architectures*, pages 34–43. Springer, 1995.
- [7] B. Lamparter, O. Böhrer, W. Effelsberg, and V. Turrau. Adaptable forward error correction for multimedia data streams. Technical report, Dept. of Computer Science, University of Mannheim, Dec. 1993. TR 93-009, <ftp://pi4.informatik.uni-mannheim.de/pub/techreports/1993/TR-93-009.ps.gz>.
- [8] S. R. McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, Computer Science Division, UC Berkeley, 1996. No. UCB/CSD-96-928, <http://http.cs.berkeley.edu/~mccanne/phd-work/>.
- [9] S. Munir. Active networks - a survey. http://www.cis.ohio-state.edu/~jain/cis788-97/active_nets, Aug. 1997.
- [10] K. Rothermel and R. Popescu-Zeletin, editors. *Mobile Agents. Proc. 1st International Workshop on Mobile Agents, Berlin*, LNCS 1219. Springer, Apr. 1997.
- [11] T. Sander and C. Tschudin. Towards mobile cryptograph. In *IEEE Symposium on Security and Privacy*, May 1998.
- [12] B. Schwartz, W. Zhou, A. Jackson, T. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks. <http://www.net-tech.bbn.com/smtpkts/smtpkts.ps.gz>, Feb. 1998.
- [13] W. Tan, E. Chang, and A. Zakhor. Realtime software implementation of scalable video codec. In *IEEE Int. Conf. on Image Processing, Lausanne, Switzerland*, volume 1, pages 985–988, Sept. 1996. <http://www-video.eecs.berkeley.edu/~dtan/icip96.ps.gz>.
- [14] D. Taubman and A. Zakhor. Multirate 3-d subband coding of video. *IEEE Trans. Image Processing*, 3(5):572–588, 1994.
- [15] D. Tennenhouse, J. Smith, D. Sincoskie, D. Wetherall, and G. Minden. A survey of active networks research. *IEEE Communications Magazine*, 35:80–86, Jan. 1997.
- [16] C. Tschudin. Open resource allocation for mobile code. In [10], pages 186–197.
- [17] C. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, University of Geneva, Switzerland, Sept. 1993. No. 2632, <ftp://cui.unige.ch/pub/tschudin>.
- [18] C. Tschudin. The messenger environment M0 – a condensed description. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, LNCS 1222, pages 149–156. Springer, Apr. 1997.
- [19] C. Tschudin. ANON: A minimal overlay network for active networks experiments. Technical report, CS Department, University of Zurich, Aug. 1998.
- [20] C. Tschudin, G. Di Marzo, M. M., and J. Harms. Welche Sicherheit für mobilen Code? In *Fachtagung Sicherheit in Informationssystemen (SIS'96), Vienna*, pages 291–307. vdf-Verlag, Mar. 1996.
- [21] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OpenArch 98, San Francisco*, Apr. 1998. <http://www.tns.lcs.mit.edu/publications/openarch98.html>.
- [22] R. Wittmann and M. Zitterbart. AMnet: Active multicast network. In A. Danthien and C. Diot, editors, “*From Multimedia Services to Network Services*”, 4th COST237 Workshop, Lisboa, LNCS 1356, pages 154–164. Springer, Dec. 1997.
- [23] N. Yeadon, A. Mauthe, D. Hutchison, and F. Garcia. QoS filters: Addressing the heterogeneity gap. In B. Butscher, M. E., and H. Pusch, editors, *Interactive Distributed Multimedia Systems, Berlin*, LNCS 1045, pages 227–244. Springer, 1996.
- [24] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New ReSerVation Protocol. *IEEE Network*, Sept. 1993.