Application Level Active Networking

Michael Fry
Atanu Ghosh

Abstract

In this paper we describe and discuss an Application Level Active Network system. This system has all the benefits of proposed Active Networks, including rapid and transparent deployment of new network services. However our system is also free of the problems of router-level Active Network deployment, such as concerns over safety and resource management.

We describe our overall architecture and its components. We then describe and discuss an implementation of the architecture in Java. We present a number of applications that have been implemented on the architecture, and indicate the benefits of our approach.

1 Introduction

Currently the deployment of new communication services is limited by the slowness of standardisation processes and the inflexibility of the communications infrastructure. Recently an approach to overcoming these problems has been proposed in the form of Active Networks (AN) [4]. AN researchers are currently proposing the deployment of protocol elements at the network router level. We believe this approach to be unrealistic. It is most unlikely that a network provider will permit the deployment of protocol code from third parties at this level. This would introduce intolerable security and safety issues, and could have a serious impact on the level of service experienced by the multiple network flows sharing the router.

We propose an Application Level Active Network (ALAN) system. Such a system consists of regular clients and servers, such as WWW [5] browsers and servers, located on the Internet or Intranet. Communication between servers and clients is enhanced by Dynamic Proxy Servers (DPS) that are located at optimal points of the end-to-end path between the server and the client. It is possible to download protocol entities onto the DPS infrastructure. These protocol entities then act as filters or enhanced protocol functionalities that improve the level of service between servers and clients. Protocol modules may be obtained from protocol servers, or more general servers such as Web servers, owned by network operators or value added service providers.

A general ALAN system consists of a number of Dynamic Proxy Servers located at strategic points of the network. The idea of special proxy machines has a well understood parallel in the current WWW infrastructure. A DPS will typically be owned by a network provider or the owner of a private Intranet. A control architecture governs the dynamic deployment of protocol modules on DPSs. There may be more than one DPS involved in a end-to-end path.

In this paper we firstly describe our overall ALAN architecture and its components, using a typical application scenario. We then describe and discuss how the architecture and its components are implemented in Java. A successful implementation of an audio streaming example indicates that the architecture is realisable. We describe further application examples that illustrate generality, and present some results that demonstrate the benefits of our approach. We conclude by discussing future issues.

Our approach enables rapid deployment of new communication services on demand without the drawbacks of current AN proposals. These services are portable across heterogeneous hardware and software platforms. The services may also be made transparent to existing servers and clients, thus avoiding extensive upgrade problems. Our system provides a platform for flexible, value added service provision.

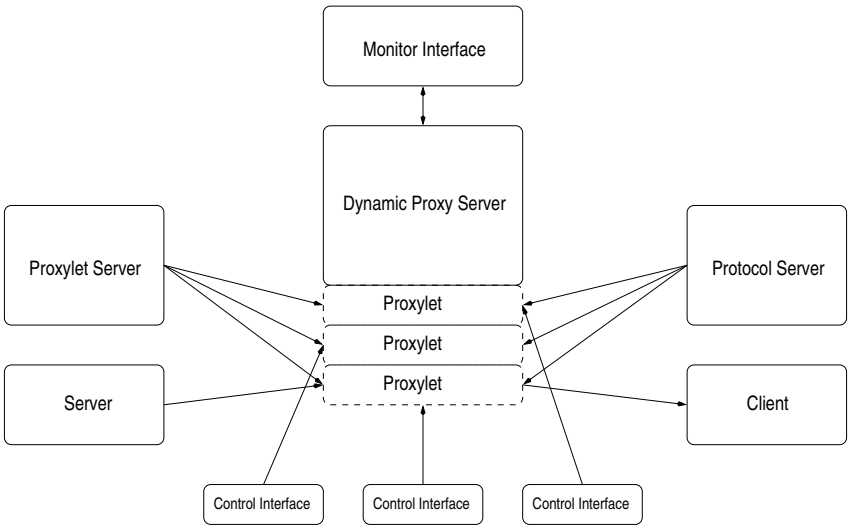

Figure 1: Architectural Overview

2 Overview - Dynamic Proxies and Dynamic Protocols

We will describe an architecture which can support Application Level Active Networking. The key ideas are as follows. In order to enhance the performance of an end to end communication *proxylets* are downloaded into Dynamic Proxy Servers which are strategically placed in the network. The end systems and the Dynamic Proxy Servers if appropriate can also down load protocol stacks from Dynamic Protocol Servers. A significant benefit of down loading protocol stacks is that stacks can then be shared. Systems developers who intend to build systems using the dynamic protocol environment will be in the position to tie together existing stacks.

A simple example scenario is as follows and is indicated in Figure 1.. A WWW server exists in the UK on which there is an audio sample. A user in Australia wishes to listen to this sample. Traditionally this sample would have been downloaded using HTTP over TCP. In our model a Dynamic Proxy Server is discovered in the UK. A proxylet is downloaded onto the Dynamic Proxy Server which pulls the audio sample from the WWW and then transcodes it to a different audio standard with perhaps higher compression than the original. The proxylet then creates Real-time Transport Protocol (RTP) packets and transmits them using UDP. The client can then receive a real-time audio stream. Further details of our actual implementation of this are given below.

3 Architectural description

3.1 Components

The system can be divided into a number of key components and concepts which are described below. Figure 1 shows how the key components fit together.

3.1.1 Dynamic Proxy Servers

These are servers which are deployed throughout the network onto which control code can be pushed to aid in end to end communications. The servers accept proxylets which are executed on behalf of a third party.

The communication with the server may be through Remote Procedure Call (RPC). In the current implementation, as Java is the implementation language, Remote Method Invocation (RMI) is used.

The Dynamic Proxy Server accepts a simple set of requests:

- Load

  A reference to a proxylet is passed to the server. A number of validation checks may be made at this stage. Some security checks may be made, such as is the load request acceptable from this particular source. The server may be heavily loaded so it may not be in the position to accept any more requests. It may be the case that proxylets can only be down loaded from prescribed servers. A service provider may only allow a controlled set of proxylets to be run on its server. If the proxylet is acceptable then it is loaded into the server. As the proxylet is loaded by reference it may be the case that the proxylet has been loaded previously and may be cached in the server, so no further downloading is necessary.

- Run

  Once the proxylet has been loaded it will be necessary to start it running. The proxylet is started using the Run request. As it is envisaged that the proxylets may take some generic form, it is possible to supply arguments with the Run request. For example a proxylet which is performing transcoding will require the names of the endpoints and which transcoding to perform. The arguments to the proxylet may also specify a particular protocol or transcoder to use. The protocols which the proxylet will use can then be down loaded from a Dynamic Protocol Server.

- Modify

  A running proxylet may require that its state is changed. For example it may be necessary to change the transcoding which is being performed.

- Stop

  Some proxylets may not naturally terminate so it may be necessary to force termination.

3.1.2 Proxylets

These are control modules which are downloaded onto the Dynamic Proxy Servers. The proxylet is a control module which binds together a number of dynamic protocol stacks. In our audio example above the proxylet uses HTTP/TCP to make a connection to a WWW server from which it down loads an audio sample. The audio coding may then be changed from PCM to GSM and then the audio is framed in RTP packets and sent onto its destination. The RTP protocol stack in this instance could be an example of a dynamic protocol stack, which may in turn be downloaded from a Dynamic Protocol Server. A proxylet may be totally self contained, with all the code required to perform the networking and filtering functions. However in order to support better code sharing and reuse of code, it may also down load part of the required protocol stacks (dynamic protocol stacks) from Dynamic Protocol Servers.

3.1.3 Dynamic Protocol Stacks

A dynamic protocol stack is a protocol stack such as a Real Time Protocol (RTP) stack. These protocol stacks reside on Dynamic Protocol Servers and are downloaded into either the end systems and/or the Dynamic Proxy Servers.

In communication systems matching protocol stacks are required to enable entities to communicate. In an attempt to simplify the interoperabilty process, protocol stacks such as RTP can be written once and placed on repositories (Dynamic Protocol Servers). When a proxylet or end system wish to interoperate, they can both adopt the same stack from a Dynamic Protocol Server.

Another possible scenario is that two ends of a communication negotiate the choice of a protocol stack. The negotiation process may reduce to one end of the communication passing a reference to the Dynamic Protocol Stack it is using to the other end of the communication.

The dynamic protocol stacks have similar advantages to shared libraries. One copy of a library is used by all applications on a system. If a modification is made to the library then only the

library needs to be replaced, not all the applications. In this model it may be possible to have single implementations of certain protocol stacks, such as RTP, available from a service provider.

If the dynamic protocol stacks adhere to a standard set of APIs then system builders will be able to use of the shelf components when building systems. The dynamic protocol stacks are an attempt to foster code reuse. Therefore enabling the fast building of systems by using existing components.

3.1.4 Dynamic Protocol Servers

These are repositories for protocol stacks, which can be used both by the Dynamic Proxy Servers and possibly end systems.

3.1.5 Proxylet Servers

The proxylets which will run on the Dynamic Proxy Servers are not directly loaded from clients but loaded by reference to proxylet Servers. Loading proxylets from servers rather than directly from clients serves two main purposes. Firstly the proxylet code can be shared if it is loaded from servers. Secondly it can be used for the validation of proxylets. It may be the case that only proxylets from trusted servers will be allowed to run on some Dynamic Proxy Servers.

3.2 Control architecture

The central core of our system is the Dynamic Proxy Server which accepts code in the form of proxylets to execute on the users behalf. Mechanisms are required for the controlling of these components. We envisage that ultimately one or more Dynamic Proxy Servers will aid in an end to end communication. In our initial model the location of the Dynamic Proxy Servers is known to the users of the system. However eventually as large numbers of Dynamic Proxy Servers are deployed such a scheme will not scale. So it is expected that the Dynamic Proxy Servers will know of each others existence, and Dynamic Proxy Servers will be discovered in an automated manner. A user wishing to make use of one or more Dynamic Proxy Servers will make a request for a Dynamic Proxy Server close to one or other end system. If we consider our earlier example of an audio sample on a WWW server in the UK, then the closest Dynamic Proxy Server to the WWW server will be discovered and used.

The Dynamic Proxy Servers have a control interface which allows remote users to connect to them in order to perform a number of control functions: causing a proxylet to be downloaded, starting the running of a proxylet, passing new arguments into a running proxylet, and stopping of a proxylet. The Dynamic Proxy Server may itself perform a number of validation checks, such as only allowing proxylets from trusted sources. It may only allow use by a set of registered users. It may not allow a proxylet to be downloaded if the Dynamic Proxy Server is currently too heavily loaded, servicing other requests.

Thus there is a control loop within which management policies and decisions can be implemented.

3.3 Downloadable entities

In order to keep the proxylets flexible they have been left as unrestricted as possible. A proxylet can contain all the code required to do any protocol processing and transformations. If appropriate the initially downloaded proxylet may contain only the control components, and may download dynamic protocol code from a Dynamic Protocol Server.

The Dynamic Proxy Servers and end systems may negotiate with each other in order to find the location of the protocol to use in a communication. In order to increase the possibility of interoperabilty all the Dynamic Proxy Servers and end systems in a communication may use the same protocol stack from the same Dynamic Protocol Server.

4 Implementation

The system that we have built thus far consists of a number of the components described above written in Java. However we feel that, while Java is useful for prototyping, the fundamental elements of our system are not dependent on Java. The following describes how the architectural components have been realised.

4.1 Dynamic Proxy Server

The Dynamic Proxy Server (DPS) is the heart of the system. It awaits requests and provides an environment for the execution of proxylets. A reference to a proxylet is passed to the DPS in the form of a Universal Resource Locator (URL). Another option may have been to have sent the proxylet as part of a request. Such a scheme would have a number of negative features, the most important of which would be that caching of proxylets by the DPS would not be possible. By placing proxylets on WWW servers we are able to leverage the ubiquity of WWW servers. Proxylets can simply be placed anywhere a WWW server exists. Another very useful feature is that the caching of proxylets is simple. The DPS could of course have implemented its own proxylet caching, however it is simpler to configure the DPS to use a local WWW cache rather as one does with a WWW browser. The DPS is therefore able to take advantage of any enhancements made to the WWW caching infrastructure.

Currently each proxylet runs in its own Java virtual machine. It would have been possible to have run all the proxylets in a single Java virtual machine, but in order to simplify the implementation, a separate process is used for each proxylet.

The Java environment supports an object oriented remote procedure call mechanism: Remote Method Invocation (RMI). This mechanism is used extensively for communication between the DPS and its clients. Two main interfaces are exported for remote access to the DPS. The first one involves the loading, running and controlling of a proxylet. The second interface provides a management interface to the DPS. The management interface can be used to retrieve the state of all the proxylets currently loaded in the DPS.

One of the actions which can be performed on a running proxylet is to stop it. When a request is made to stop a proxylet its stop method is called and if, after thirty seconds it still persists, the Java virtual machine is killed. This mechanism seemed simpler to implement than to attempt to identify all threads associated with a particular proxylet. There are certain conditions where it is not possible to kill a thread. One example of this is when a thread is waiting for data from the network (it's not clear if this is a feature or a bug). We also had concerns that proxylets running together may inadvertently or maliciously interfere with other proxylets in the same Java virtual machine.

A number of security and authentication checks are made before a connection is allowed to the DPS. These checks are enumerated below. As well as the security checks, hooks are in place in the DPS to disallow the loading of a proxylet if the machine on which the DPS is running is too heavily loaded.

4.2 Proxylets

A proxylet is dynamic code which is downloaded and run on a DPS. A proxylet implements an interface, much in the same way that an applet running in a WWW browser implements an interface. Initially we had intended to limit the operations that a proxylet was able to perform. However it seemed to provide much more flexibility to allow a proxylet access to all the classes in the Java Development Kit (JDK). A proxylet can therefore have access to all the functionality that is available in the JDK.

A proxylet physically exists as a collection of class files. A class file is produced as a result of the Java compilation process. These class files contain machine independent byte codes. The class files are bundled together in single Java archives (JAR files). Each proxylet exists in a single

Jar file. As WWW servers are already repositories of arbitrary data, we decided to place the Jar files which contained our proxylets on WWW servers.

In the architecture of the ALAN system we defined Dynamic Protocol Servers. These servers were envisaged to be WWW servers which kept Protocol elements rather like proxylets. It was intended that a proxylet would be able to draw in protocol elements such as RTP code or various audio codecs. This partitioning was intended so not only proxylets but other entities would be able to share the same code. For example a lot of code is currently shared between the audio transcoder proxylet which has been written and our audio tool YAAT [2]. We had intended to extract this shared code and place it on a protocol server. If we had done this then any time a new audio codec is written it could be placed on a Protocol server and both the proxylet and the audio tool YAAT could dynamically load the new code. This part of the ALAN infrastructure has not yet been implemented.

We discuss below some security ramifications. Currently we only envisage proxylets being used in network transactions. The current generation of proxylets that we have written have full access to all the network classes. It will perhaps be necessary in the future to control via the security manager the network bandwidth used by a particular proxylet. If the amount of bandwidth a proxylet requires is known in advance, this information can be used to determine if enough bandwidth is available on a DPS to allow a particular proxylet to loaded and run.

4.3 Control Interface

A simple generic control interface has been written which is used to initiate the download of proxylets into a DPS. That is, other than for our WWW-based example described below, at present proxylets are loaded manually. The interface also allows the client applet to be started and stopped.

Another operation is also supported which is to pass new parameters to the proxylet while it is running. This feature is used in our audio transcoder proxylet to change the encoding or add redundancy during playout. It is envisaged that specialised control interfaces would be developed for some proxylets. In the case of the audio transcoder proxylet, changing an audio parameter requires typing in a special string which is parsed by the proxylet. For ease of use a special interface could be developed.

4.4 Monitor Interface

A monitoring interface has also been developed which is used to observe the proxylets running on a DPS. Currently this interface only allows observation of the state of proxylets on a particular DPS. Enough information is presented for a Control Interface to be started to take control of a running proxylet. Eventually it is intended that the Monitor Interface will support taking control of any running proxylets.

4.5 Security

We have previously been critical of the pure Active Network concept on the basis of security and safety. Clearly, if arbitrary proxylets can be downloaded and run on foreign machines running DPSs, security is still a major concern. We intend to address this further in our ongoing work. However we feel that, within an application layer solution using a control protocol, the problem is more tractable. In our current implementation the following security hooks are in place:

- Only trusted hosts are allowed to submit requests to a DPS.
- Proxylets can only be downloaded from trusted WWW servers.
- Before a proxylet is run it is verified that it is correctly signed. Jar files are used both for applications and applets. It is possible to add a digital signature to a jar file in order to authenticate the creator of the jar file. This scheme is in place to allow trusted proxylets

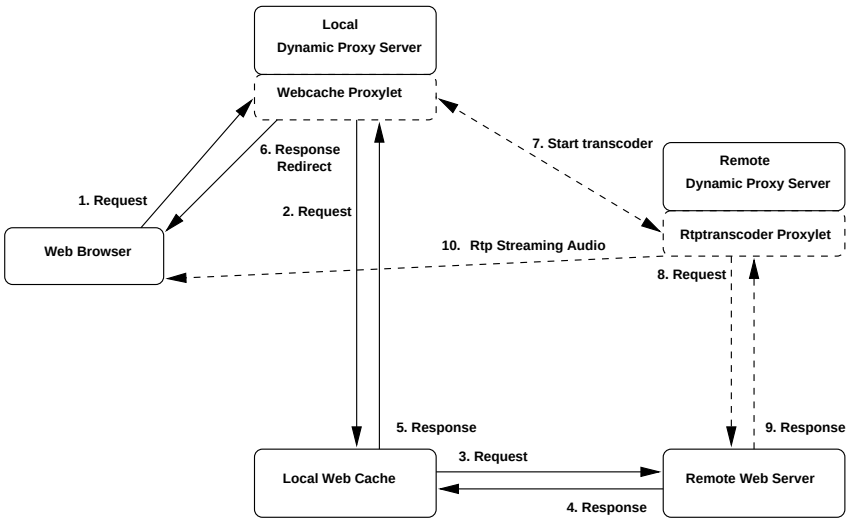

Figure 2: Web Audio Transcoding

to perform normally untrusted operations, such as accessing files on a local file system. We intend to use the same scheme to authenticate our proxylets, so only trusted proxylets will be allowed to run on a DPS. Unfortunately, at this time we are at a transition point between Java version 1.1.6 and version 1.2. Between the two versions of Java the signing mechanism is changing so we have decided to await a stable 1.2 release before implementing signing.

It is also undesirable that a proxylet is able to read or modify local files on the machine of which a DPS is running. Java has a mechanism for adding ones own security manager to the run time system. We intend to add a security manager to disallow certain local operations.

4.6 Discovery

Currently the locations of our DPSs are known. We therefore make explicit connections to a particular DPS when we wish to start a proxylet. In the future it is intended that that DPSs communicate with each other, and that when a request is made to run a proxylet it is loaded and run on the most appropriate DPS.

5 Experiments

We have performed a number of experiments with the infrastructure that we have in place, using different applications or application services.

5.1 WWW streaming audio

We have previously reported on an audio transcoder which takes audio samples from a WWW server transmitted using HTTP/TCP and converts to a RTP stream [2]. We decided to try to seamlessly integrate this work into a WWW browser and WWW server interaction. A key challenge was to make the system transparent to both server and browser. The solution lay in the creation and use of a *webcache proxylet*.

Most WWW browsers have the ability to be configured such that all transactions are performed through a local cache. Rather than have all HTTP requests go directly to the site which is pointed to by a requested URL, they first go to a local cache. If the URL is cached locally it is returned from the local cache. If the URL is not present on the local cache it makes the request to the

remote site. If the URL is of a type that can be cached it is then cached for future requests. The setting in the WWW browser to point to a cache is typically called the proxy variable.

Our configuration was thus fairly simple. Figure 2 shows the interactions involved. A DPS is run at the site where the user is WWW browsing. A webcache proxylet is started on the local DPS and the proxy variable in the users browser is set to the webcache proxylet DPS machine. The webcache proxylet does not actually perform any caching, but this scheme allows the webcache proxylet to be transparently in the path of HTTP requests.

A HTTP request involves a request for a page on a WWW server. Preceding the requested page in the returned stream is a mime content type defining the content of the stream. This allows the browser to correctly render the incoming stream. A common content type is "text/html" which the WWW browser renders as plain text. Another content type is "audio/basic", which causes the browser to download all the content into a file, and then attempt to start an audio program on the local file.

The webcache proxylet is in the position to observe the content types of the requests flying past. The webcache proxylet has a table of mime content types on which it is able to perform special operations. If a mime content type has no table entry, then the webcache proxylet simply relays the request to the real local WWW cache machine. These are content types to which the system can add no value.

However, in our experiment, if the webcache proxylet observes that the content type being returned is type "audio/basic", then it is in the position to perform special processing. The webcache proxylet should attempt to identify a DPS which is close to the source of the audio, i.e. the WWW server on which the audio sample resides. This dynamic component of the operation has not yet been implemented. In the current version of the webcache proxylet there is prior knowledge of the location of appropriate DPSs. Once an appropriate DPS is identified an audio transcoder proxylet is started by the webcache proxylet on the newly discovered DPS.

The audio transcoder proxylet is started with the location of client browser as well as the original audio URL. The audio transcoder makes a standard HTTP request for the audio sample, then converts the incoming stream to a RTP stream which is sent on to the users machine.

The WWW browser it not expecting a RTP stream so some special actions have to take place. The webcache proxylet can perform one of two actions. It can send back a new content type "audio/yaat". If the WWW browser has previously been correctly configured this will cause an RTP compliant audio tool to be started. The tool started can either be one of the standard tools such as vat [9], or rat [10], or yaat: our own Java audio tool which we had written as part of our previous experiments[2]. However this solution is not totally flexible or transparent, since it requires the client's local environment to be configured to start an audio tool. We have thus developed an alternative solution

Rather than send back a new content type, the webcache proxylet can instead send a URL redirect to the WWW browser. This causes the users WWW browser to go to a different WWW page. We have a page configured which contains a version of our audio tool configured as an applet. In this case no preconfiguration is necessary, but the WWW browser must be capable of running applets. It is also possible on the WWW page which contains the yaat audio applet, to have a control applet which can be used to change the audio coding and level of redundancy used by the audio transcoder. We are currently able to start a control application manually to manipulate the audio transcoder, but at the time of writing this process has not been automated.

5.2 WWW compression

The webcache proxylet allows WWW-based protocol conversions and enhancements to be made in the network, transparently to the end systems. In the case of the audio streaming example it is possible to deploy a audio player applet without any special configuration of the WWW browser.

The webcache proxylet can also be used to perform special processing on text pages (Figure 3). When the webcache proxylet sees a page of content type "text/html" it arranges for a decompresser to run locally and a compressor proxylet to run remotely. The compression and decompression

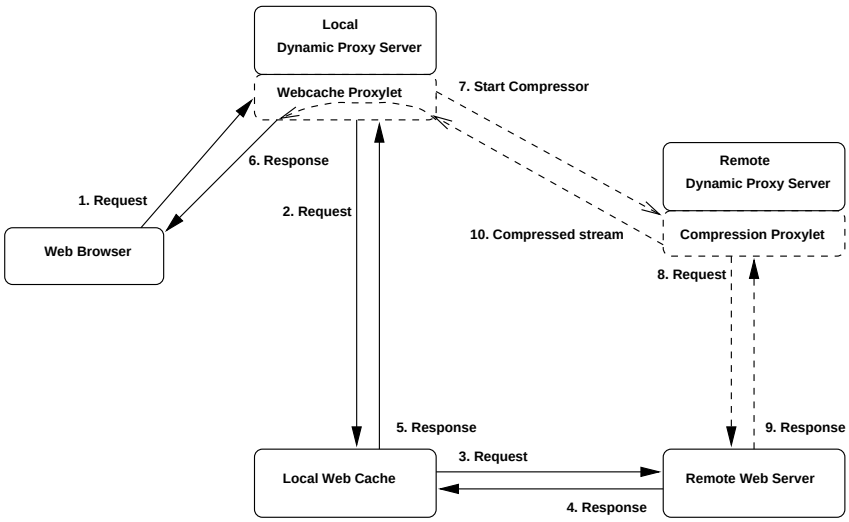

Figure 3: Web Compression

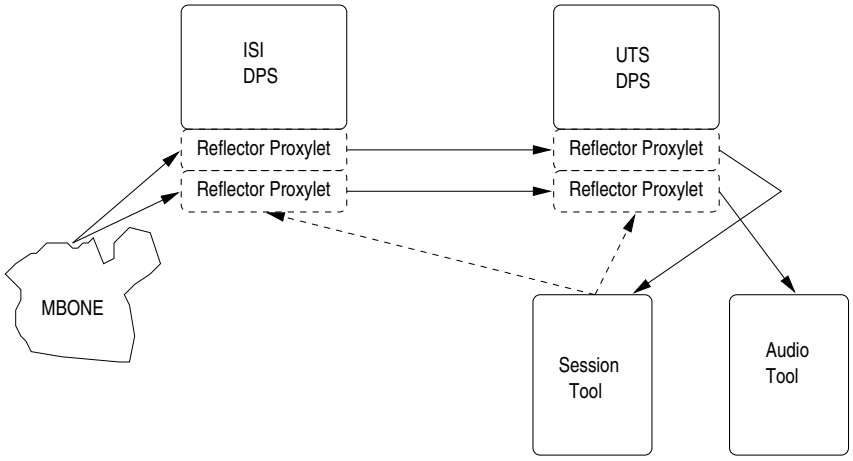

Figure 4: Tunnelling from the Mbone

stages are totally transparent to the WWW browser and WWW server. For congested links this compression enhances the performance of HTTP connections.

5.3 Multicast reflector

At the time of writing Australia is not connected to the global Mbone [7]. During the recent Internet Engineering Task Force (IETF) meeting in Los Angeles we wanted to view some of the sessions. We had been generously allowed to have an account on an ISI machine at MIT on which we occasionally run a DPS. In order to demonstrate the simplicity of our proxylet infrastructure a simple Java application, SAP, was written which could receive and process session description protocol (SDP [8]) packets. These SDP packets describe multicast sessions such as the IETF sessions which were taking place.

We required a mechanism to join multicast sessions. A simple reflector proxylet was written which would join a multicast group. Thence all packets received by the proxylet would be sent unicast to another reflector proxylet which would re-multicast the packets locally. A DPS was run at ISI and a DPS was run locally at UTS. One multicast channel is used to disseminate session information. The DPS at ISI had a reflector running on it which was listening on the session

description channel. These packets were then forwarded unicast to UTS where the peer reflector was multicasting them locally.

Therefore it was possible to see all the advertised sessions on the Mbone. We were in effect tunnelling session information from ISI to UTS, giving us access to all the session information. The next stage was to use the SAP program to join particular sessions. Thus when a session is selected the SAP tool, as well as offering the option of starting local tools for the video, audio and other media, also offers the option of starting a reflector for that session. Using the SAP program it is possible to select a session and arrange for a pair of reflectors to be started at ISI and UTS. It was therefore easy to listen to sessions we were interested in, once the original pair of reflectors had been started.

In Figure 4 we see one pair of reflector proxylets forwarding the session information stream to the UTS network. A stream such as an audio stream can be selected within the SAP tool, which causes another pair of reflector proxylets to be loaded for this stream. The SAP tool has knowledge of the location of the two DPSs in order to start each new pair of reflector proxylets.

Our simple implementation has a number of shortcomings, such as lack of feedback information into the Mbone, limited forwarding of session information, and inability to stop a stream automatically. However it would not be difficult to correct these shortcomings. It would also be possible to add enhancement services. For example, in this version of the reflector proxylet we send the whole stream unmodified. It would not be difficult to add code to transcode the audio and video streams, to reduce the bandwidth or add redundancy, as we did with the WWW streaming example.

As we report below, the file containing the reflector proxylet source code is only 170 lines and took a few hours to write. The program to parse the SDP packets took a little longer but was not strictly necessary as a simple plugin could have been written for an existing tool such SDR [12], to start the reflector pair.

5.4 Tcpbridge

We have often made the observation that the path chosen through the network by routing protocols is not always the one offering the least latency. A classic example of this, discovered experimentally during SIGCOMM97 in Cannes, is that the direct network path from southern France to UTS in Australia can be very poor. Far better response can be achieved by first logging into a machine at University College London (UCL) and then logging into UTS.

It seemed that some application level routing would improve things. We therefore wrote a simple tcpbridge proxylet, comprising a source files of 86 lines. The proxylet accepts connections on a certain port and then in turn makes a connection to another host and port. All data in both directions is simply relayed through the bridge.

This proxylet can be viewed as an application level generalisation of the snoop solution to wireless tails. It isolates troublesome segments of the net with their own, dedicated TCP connections. This prevents known problems that can occur when using purely end-to-end error correction and congestion control.

6 Results

6.1 Performance measures

A potential issue with application level solutions is performance. We now present results that indicate the impact of our approach on bandwidth and latency.

6.1.1 Streaming throughput

We have previously reported results for our audio transcoder [2]. These are reiterated here. The results were obtained using a Sun Ultra 1 machine running Solaris 2.5.1 as the DPS.

Bytes	No Proxylets	Proxylets (Compression)	Connectivity
1	2.6s	5.7s	Ethernet (10Mb/s)
5073934	20.12s	21.52s	Ethernet (10Mb/s)
5073934	25.83s	19.85s	Wavelan (2Mb/s)
5073934	13:8min	3:42min	Wide Area

Table 1: Text Retrieval Times

The main computational concern was that an audio sample could be coded or decoded in less time than the sample represented. The smallest time sample supported is a 20ms sample. For PCM audio samples the transcoder does nothing. In the ADPCM case the transcoders convert from PCM to ADPCM or ADPCM to PCM. To decode a 20ms ADPCM audio sample on a Sun Ultra 1 takes between 1-2ms. This is well inside the 20ms required. The worst case is where 20ms ADPCM audio samples are created with two levels of audio samples. This process takes 4-5ms. The time taken for two levels of redundancy for 20ms samples and the transmission of the packets including the packetisation is ~5ms. So using the ADPCM codec with two levels of redundancy and a sample size of 20ms a packet needs to be transmitted every 20ms. It actually takes ~5ms, which leaves ~15ms available. So for a simple transcoder like ADPCM there is plenty of time.

This analysis was further confirmed by the fact that the audio sample was quite comprehensible to the listener. A wide area experiment, conducted between Australia and the UK, further demonstrated the viability of this approach, and also the value of dynamically chosen options such as redundancy and sample size.

6.1.2 Latency and cost

Our model involves proxylets being transfered across the network to be run on DPSs. A certain amount of overhead is involved in the actual transfer of a proxylet and the time required for a DPS to start running a new Java VM.

A pathological case may be that an audio sample is requested from the UK to Australia. The transcoder proxylet is actually located in Australia and is larger than the original audio sample. Therefore before playout can begin a file transfer of the proxylet from Australia to the UK takes place before a smaller amount of information is transferred in the opposite direction. However, as mentioned above the DPSs can be configured to use local WWW caches. Thus once a proxylet has been downloaded it should not need to be downloaded again unless it has changed.

In order to get an initial feel for the overheads of the ALAN infrastructure and the possible savings involved, we performed some experiments with the compression proxylet. As mentioned above the webcache proxylet is able to download a compression proxylet to a DPS close to the target WWW server. The compression proxylet retrieves the requested URL and compresses it. The compressed stream is then sent back to the webcache proxylet which decompresses it and then delivers back to the WWW browser. The GZIP compression algorithm is used.

We wrote a small Java program to retrieve the contents of a URL. We did not wish to incur the extra overhead of a standard WWW browser attempting to parse the incoming stream. It was also easier to time a program which exited when the whole URL had arrived. The experiments were carried out using the proxylet infrastructure, and also without it. The same WWW cache was used in both instances. When using our local network each experiment was repeated twenty times and the average measurement recorded. Two files were used: one containing a single byte of text, and the other the entire King James' Bible (5073934 bytes).

The results of the experiments are given in Table 1. It can be seen that the overhead of using the proxylet infrastructure seems to be around three seconds when using a single byte file. For tests performed locally on a switched 10Mb/s Eternet we see no saving in transfer time. This is because the bottleneck in this case is the compression time.

In our set-up the compression time for the Bible is around 10 seconds (the decompression time as expected was an order of magnitude less), which gives a compression throughput of around

Operating System	Processor	JDK	Time (ms)
Solaris 2.6	Two 200MHz UltraSparc's	JDK 1.1.5	8770
Solaris 2.6	150MHz SparcStation 20	JDK 1.1.5	9582
Solaris 2.6	167MHz Ultra 1	JDK 1.1.5	10448
Dec Unix 4.0B	AlphaStation 250 4/266	JDK 1.1.5	13661
Windows NT Workstation	166MMX Intel	JDK1.1.6	14388
Linux 2.0.33	166MMX Intel	Linux_JDK_1.1.5_v5	17893

Table 2: Compression Times

4Mb/s. We hypothesised that for links with a lower bandwidth than this, using the compression proxylet would be beneficial. We therefore ran the same experiment over a wireless link (Wavelan) which has a bandwidth of 2Mb/s. It can be seen that there is a five second reduction in transmission time using the compression proxylet. We believe the faster time is due to not having to transmit the whole umcompressed bible over the physical network link. (The transmission time of the bible is actually less when going over the Wavelan interface than when using the Ethernet. In the case of the Wavelan experiment, the mobile host has the DPS running the webcache proxylet and hence the decompression proxylet.)

To further test our hypothesis we tried running a compression proxylet at University College London and a decompression proxylet at the University of Technology, Sydney. We transfered a whole bible in this way a few times. We in fact found a wide variance in our measurements, which we believe are reflective of transient changes in network congestion across the global Internet. The measurement shown in the final line of Table 1 is about the average of our measurements. We can see that the retrieval time seems to be substantially improved by the use of compression using the DPS infrastructure. It is approximately 29% of the uncompressed latency, which is a close match to the compression ratio (below).

Our measurements strongly suggest the benefits available to the end user. There are also of course potential cost savings. The bible file was compressed from 5073934 bytes to 1449540 bytes, which is a compression ration of 3.5 (or 28.5%). Under a incoming volume based charging regime (such as operates in Australia) the compression proxylet can significantly reduce network charges.

Finally, the cross-over point for deriving benefit from our compression example we estimated at about 4Mb/s. This is clearly determined by the CPU speed available at the compressor. We experimented with just compressing the bible file on various architectures that we had available. The results are in Table 2. Obviously a more powerful engine at the compressor will drive down the cross-over point.

6.2 Development and deployability

Table 3 gives some metrics of the proxylets that we have developed. The source code lines quoted are all the lines in the source files, including blank lines and comments. All the proxylets apart from the transcoder were written from scratch once the ALAN infrastructure had been developed. The transcoder proxylet was built from code that had been written previously, so there are remnants of redundant code and the Jar file is much larger than it should be. Also, the time for development of the transcoder is given as only one day. However, as transcoder code already existed it was just a question of packaging it as a proxylet. The webcache proxylet has been evolving over time, so it is difficult to estimate how much time has been spent in total. The other proxylets were written in a single day or less, and have required little if any modification.

For the majority of the simple proxylets written the development time can be seen to have been very short. The deployment time has been even shorter. All that is required is to compile the proxylet package into a Jar file and then place it on a WWW server. This whole operation can be performed by a few simple commands in a makefile. This is in fact how we install our proxylets.

Proxylet	Jar size Bytes	Source code Lines	Development time Days
Webcache	23888	938	
Transcoder	52307	3898	1
Compressor	2866	233	1
Tcpbridge	2559	86	1
Reflector	2841	170	1

Table 3: Proxylet metrics

6.2.1 Ease of use

The proxylets typically are short lived and exist inside the ALAN infrastructure. Versioning and update problems do not exist in the same way that they do with other software. If a proxylet exists on a single WWW server then an update is simply performed by changing this one copy. As a DPS can take advantage of the WWW caching infrastructure, once a local copy of a proxylet exists in the local cache there should be no need to ever retrieve it again unless it changes. During the development and debugging phase of the proxylets it has been trivial to disseminate changes by just restarting a proxylet.

Once a scalable, hierarchal WWW cache strategy is in place there should not be any reason to ever have a proxylet exist on more than one WWW server. The WWW servers on which proxylets reside will of course need to be highly available. Network problems which stop a proxylet being accessed will of course always be a problem.

7 Related work

Our work is significantly different to that embedded in existing WWW-based streaming tools. For example, the RealAudio tool from Progressive Networks supports streaming audio. Special purpose Audio servers and client players are used for the playout of the audio. Code cannot be pushed and pulled transparently, and support for adaptation is limited.

In the introduction we compared our approach to that of the Active Networks community. There is valuable work being carried out by that community, e.g. the ANTS toolkit, which addresses mobile network code deployment. However this work still has all the drawbacks associated with router-level deployment. Our ALAN approach avoids these problems by limiting deployment at the application level in carefully managed DPS machines. The control architecture should address security and safety issues and will not impact normal resource sharing. What we are proposing is an application level, end-to-end solution, which we have shown to be both efficient and realistically deployable.

Our work has some similarities with transcoder work where, at tails of the network, a high bandwidth stream needs to be transformed to a lower bandwidth stream. The Dynamic Proxy Server could be used at the intersections of the high/low bandwidth networks.

8 Conclusions and future work

We have proposed a novel architecture and mechanisms for the achievement of Application Level Active Networks. We have demonstrated the feasibility of our proposals via experimental implementation.

Future work will address the experimental implementation of the entire system. This will focus on the realisation of the overall control architecture and fully downloadable proxylets and protocol stacks. Some issues to be explored further include safety, location and discovery of Dynamic Proxy Servers, dynamic protocol stacks, and caching.

9 Acknowledgements

This work is funded by British Telecom Laboratories. Many thanks to Ian Marshall for the fruitful discussions and suggestions. We would also like to thank Glen MacLarty for writing the webcache proxylet.

References

[1] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith, "Active Bridging", SIGCOMM Vol 27 Number 4, pp 101-111. October 97.

[2] A. Ghosh and M. Fry, "JavaRadio: an application level active network", Third International Workshop on High Performance Protocol Architectures(HIPPARCH '97), June 97.

[3] Ken Arnold and James Gosling, The Java Programming Language. *Reading, Massachusetts: Addison Wesley*, 1996.

[4] David Tennenhouse, D.J. Wetherall Towards an Active Network Architecture. *Computer Communication Review,* Vol. 26, No. 2, April 1996, pp. 5-18.

[5] Tim Berners-Lee, The world-wide web initiative. *Proceedings of the International Networking Conference (INET), (San Francisco, California), pp. DBC-2 - DBC-4, Internet Society*, Aug. 1993.

[6] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RFC 1889 *RTP: A Transport Protocol for Real-Time Applications*, 01/25/1996.

[7] V. Kumar, *"MBone: Interactive Multimedia On The Internet"*. Macmillan Publishing (Simon & Schuster), 1995.

[8] M. Handley, V. Jacobson, RFC 2327 *SDP: Session Description Protocol*, 04/14/1998.

[9] Stephen Casner and Stephen Deering, "First IETF Internet audiocast," *ACM Computer Communication Review*, vol. 22, pp. 92-97, July 1992.

[10] V.Hardman, A.Sasse, I.Kouvelas Successful Multi-party Audio Communication over the Internet, *Communications of the ACM*, 1997.

[11] <URL:http://jeeves.javasoft.com/>

[12] M. Handley, SDR Manual page.