Application Level Active Networking UCL HIPPARCH Project Deliverable D4

Michael Fry, Atanu Ghosh and Jon Crowcroft
University of Technology, Sydney
Building 4, Thomas Street
PO Box 123
NSW 2007
Australia and
Department of Computer Science, UCL
Gower Street
London, WC1E 6BT
UK

May 18, 1999

Abstract

In this document we describe and discuss an Application Level Active Network system. This system provides the benefits of proposed Active Networks, including rapid and transparent deployment of new network services. However our system is also relatively 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) [11]. 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 [4] 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. There may be more than one DPS involved in a end-to-end path. 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.

The idea of special proxy machines has a well understood parallel in the current WWW caching infrastructure. Dynamic Proxy Servers, and the associated proxylet and protocol servers, will typically be owned by a network provider or the owner of a private Intranet. This constraint, we believe, makes security issues more tractable. Acceptable security can be achieved using known mechanisms.

In this document 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.

The main contribution of our approach is to enable 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. We have developed our proposal to proof-of-concept level. While our approach may be generalisable to permit dynamic deployment of active elements to clients and servers, we have chosen initially to focus on services that can be made transparent to existing servers and clients. This has many benefits, such as helping to solve extensive upgrade problems.

Our system provides a platform for flexible, value added service provision. However our work is incomplete. We have not yet developed dynamic binding of protocol elements. We need to develop and test a scalable system architecture. This will permit the dynamic discovery and location of DPSs. Likewise further work is required to address the manageability and robustness of the system, including specification of appropriate security mechanisms.

2 Architectural Description

2.1 Overview

We describe an architecture which can support Application Level Active Networking. The system can be divided into a number of key components and concepts which are further described below. Figure 1 shows how the components fit together. 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. These proxylets are obtained from Proxylet Servers. We also intend that Dynamic Proxy Servers can, if appropriate, also download protocol stacks or protocol elements from Dynamic Protocol Servers. However this aspect has yet to be implemented. A significant benefit of downloading proxylets and/or protocol stacks from common servers is that code can shared. Systems developers who build systems using this environment can bind together existing code modules.

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. Without enhancements to the server and browser this sample is fully downloaded using HTTP over TCP before being played out. In our approach a Dynamic Proxy Server is discovered in the UK. A proxylet is downloaded onto the Dynamic Proxy Server, which then pulls the audio sample from the WWW server. The proxylet transcodes

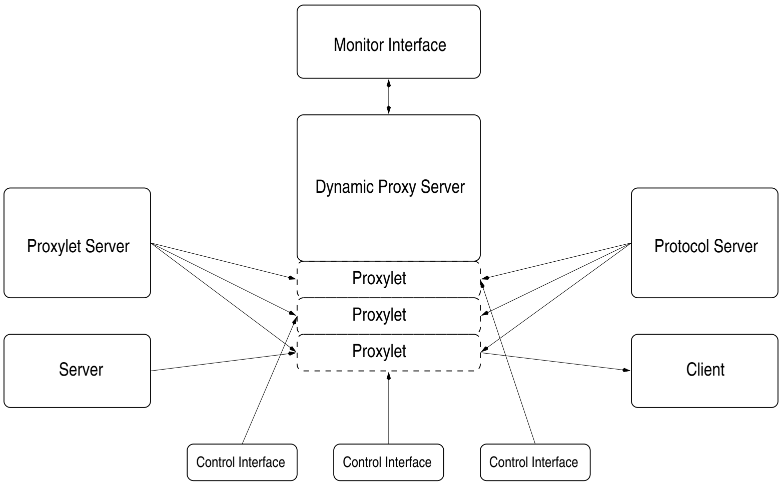

Figure 1: Architectural Overview

the sample to a different audio format with higher compression than the original. The proxylet then creates Real-time Transport Protocol (RTP) packets and transmits them using the UDP transport protocol. The client can then receive a real-time audio stream. Further details of our actual implementation of this proxylet are given below.

2.1.1 Dynamic Proxy Servers

Dynamic Proxy Servers (DPSs) accept proxylets which are executed on behalf of third parties. Communication with a DPS is through Remote Procedure Call (RPC). Since Java is the current implementation language, Remote Method Invocation (RMI) is used. The Dynamic Proxy Server accepts a simple set of requests. The DPS exports a RMI interface for these requests and clients call the exported methods using RMI. The requests are as follows.

- Load

  A reference to a proxylet is passed to the server. A number of validation checks may be made at this stage. 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 downloaded 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 downloaded from a Dynamic Protocol Server (discussed below).

- 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.

2.1.2 Proxylets

These are control modules which are downloaded onto the Dynamic Proxy 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 common servers serves two main purposes. Firstly the proxylet code can be shared if it is loaded from servers. Secondly it can be used to validate that proxylets are sourced from trusted servers.

2.2 Downloadable Entities

A proxylet may be totally self contained, with all the code required to perform the networking and filtering functions. Alternatively the proxylet may be 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.

At the time of writing we have not yet implemented dynamic protocol stacks. In the following section we simply discuss our proposed approach and some related issues.

2.2.1 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 may in principle be downloaded into either the end systems and/or the Dynamic Proxy Servers.

In communication systems matching protocol stacks are required to enable entities to interoperate. To simplify the interoperabilty process, protocol stacks such as RTP could 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 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 off the shelf components when building systems. However it may be extremely difficult to develop generic APIs for all protocols. Furthermore there is a known trade-off between modularisation of protocol layers or functions and protocol performance. This is due to overheads of modularisation and to interaction between features of modularised components. These issues need to be explored through further work on dynamic protocol stacks.

3 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.

3.1 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). 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 implement 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 be possible to run all the proxylets in a single Java virtual machine, but to simplify the initial implementation we decided to use a separate process 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 as described above. 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.

3.2 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 more flexible to allow a proxylet access to all the classes in the Java Development Kit (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 place the Jar files which contain our proxylets on WWW servers. Thus a proxylet can simply be referenced via a URL.

We discuss below some security issues. 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 load and run.

In our current implementation there is a control loop within which management policies and decisions can be implemented.

We have not currently defined interfaces or APIs that enable proxylets to cooperate. In this initial phase we export custom interfaces for this purpose. Generic interface requirements may become apparent after some experience has been gained in developing cooperating proxylets.

4 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. Typically one or more Dynamic Proxy Servers will aid in an end to end communication. Currently the locations of our DPSs are known. We therefore make explicit RMI connections to a particular DPS when we wish to start a proxylet. This explicit information is provided via the Control Interface (described below). However as large numbers of Dynamic Proxy Servers are deployed such a scheme will not scale. The Dynamic Proxy Servers will need to know of each others existence, and Dynamic Proxy Servers will be discovered in an automated manner. For example, 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 scenario 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 DPS location problem requires further work. However as an initial approach, providers such as ISPs could locate their Dynamic Proxy Servers, and have them exchange information, in a similar manner to WWW cache proxies. Nonetheless a more generic DPS location strategy will be more challenging.

4.1 Control Interface

A simple generic control interface is used to initiate the download of proxylets into a DPS. That is, other than for our WWW-based examples described below, at present proxylets are loaded manually.

Another operation is supported 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 will be developed for some proxylets to enable user interaction.

4.2 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. 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.

4.3 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, for our application layer solution, the problem is more tractable. In particular we initially assume that DPSs and Proxylet Servers are within the domain of a single owner, e.g. a network provider and/or ISP. A DPS will not allow proxylets to be executed that originate from other arbitrary sources. With this constraint we believe the security problem to be soluble using existing mechanisms.

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. It is possible to add a digital signature to a Jar file in order to authenticate the creator of the file. This scheme is in place to allow trusted proxylets to perform normally untrusted operations, such as accessing files on a DPS 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 read or modify local files on the machine on 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.

In the future we intend also to relax the constraint that DPSs only execute proxylets from trusted sources. This will require further research into security and safety issues.

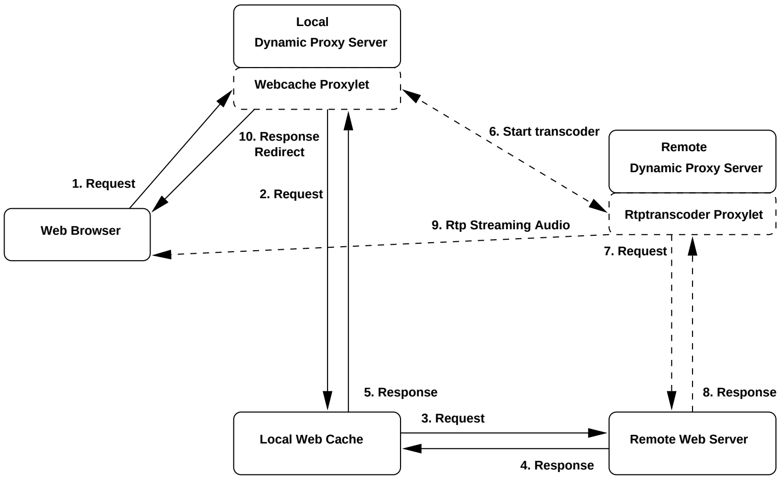

Figure 2: Web Audio Transcoding

5 Experiments

We have performed a number of experiments with our infrastructure 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 [6]. 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 server which is pointed to by a requested URL, they first go to a local cache. If the page 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 server. If the page 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 idea 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.

In the following description the numbers in parentheses are references to the interactions in Figure 2. A HTTP request involves a request for a page on a WWW server (1-3). Preceding the requested page in the returned stream is a mime content type defining the content of the stream (4-5). 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 all responses flying past (5). 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 response to the browser. 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 (6).

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 (7), then converts the incoming stream (8) to a RTP stream which is sent on to the users machine (9).

The WWW browser is 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, say "audio/yaat". If the WWW browser has previously been appropriately configured, this will cause an RTP compliant audio tool to be started. The tool started can either

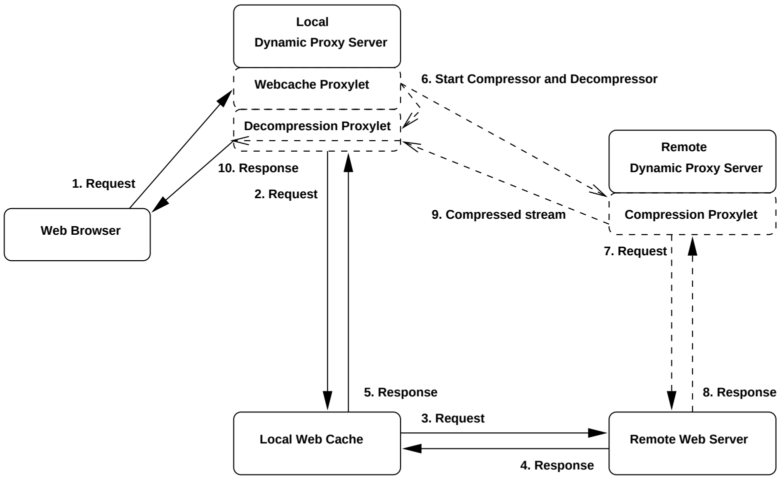

Figure 3: Web Compression

be one of the standard tools such as vat [5], or rat [8], or yaat: our own Java audio tool which we had written as part of our previous experiments[6]. 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 (10). This causes the users WWW browser to go to a different WWW page. This page contains a version of our audio tool configured as a Java applet, which is thus automatically loaded. In this case no browser preconfiguration is necessary. 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.

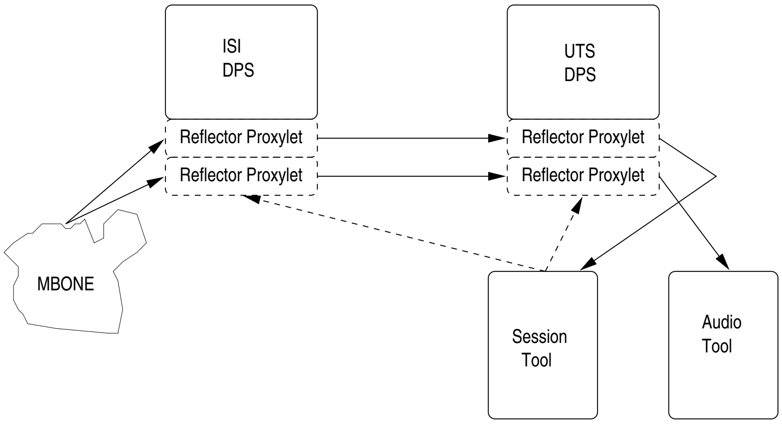

Figure 4: Tunnelling from the Mbone

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

5.3 Multicast Reflector

At the time of writing Australia is not connected to the global Mbone [9]. We required a mechanism to join multicast sessions. In order to demonstrate the simplicity of our proxylet infrastructure we first developed a simple Java Session Tool which could receive and process Session Description Protocol (SDP [7]) packets. These SDP packets describe current multicast sessions on the Mbone.

Previously, we had generously been allowed access to a machine at the Information Sciences Institute (ISI) located at the Massachussets Institute of Technology, on which we occasionally ran a DPS. We thus developed a reflector proxylet to execute at the ISI DPS. This reflector listens on the session description channel and forwards packets unicast to the University of Technology, Sydney (UTS), where a peer reflector multicasts them locally. It is thus possible, via the reflector and the session tool, to observe all advertised Mbone sessions. We are tunnelling session information from ISI to UTS.

The next stage was to use the session tool to join sessions. Thus when a session is selected the tool, as well as offering the option of starting local applications for the video, audio and other media, also offers the option of starting a reflector for that session. If this option is selected a further pair of reflectors are started at ISI and UTS to tunnel the actual session payloads.

Figure 4 illustrates the operation of the system. We see one pair of reflector proxylets forwarding the session information stream to the UTS network. Selection of (in this case) an audio stream through the session tool causes another pair of reflector proxylets to be loaded for this stream. In practice this proxylet-based enhancement has enabled us to observe audio and video Mbone conferences with no apparent lack of quality vis-a-vis a regular Mbone feed.

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 transcoding of audio and video streams, e.g. to reduce bandwidth or add redundancy, as we do with the WWW streaming proxylet.

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 typical example of this, discovered empirically, 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.

We hypothesised that some application level routing can improve latency. We therefore wrote a simple tcpbridge proxylet, comprising a source file 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.

The proxylet could also be modified to provide a TCP-splicing function as is quite commonly required in proxy machines such as caches and firewalls [10].

6 Distributed Protocol Demonstrators

In this section, we describe two extended uses of the ALAN Architecture to provide distributed protocol support for applications that require multicast. The first accomodates reliable delivery of data to receivers over heterogeenous network paths. The second accomodates delivery of real-time (or near real-time) audio (or video) to receivers over paths that have different data rates.

These two goals could in principle be achieved by extending router functionality, or by use of multiple multicast channels. Neither of these approaches would scale well today or indeed in future Internetworks.

6.1 RLC

In UCL D1 we described the use of multicast in a web based scenario. In the demonstrator in 1998, we showed this operating in hand crafted, and automatically generated versions, as described in UCL D3. The protocol is described in detail in UCL D2, and makes use of layered FEC to deliver data reliabilyi, as well as providing efficient TCP-like congestion avoidance. In doing so, the protocol accomodates seperate receiver rates.

To deploy the protocol effectively, we must avoid any radical changes to web servers and web broswers. One approach is to deploy the client side as a plugin (or applet). However, since the server side *must* be deployed as a proxy (due to the lack of common support for servlets, for good security reasons), it seems eminently reasonable to implement both the client and server sides as proxylets in ALAN, and then have users direct their browsers at the appropriate DPS.

Conveniently, the last implementation of RLC was written in Java [1].

It is thus relatively easy to deploy this.

[1] `http://www.cs.ucl.ac.uk/external/L.Vicisano/rlc/`

6.2 Self-Organising Transcoders

There are very many streaming servers in the Internet now - With the advent of MP3 and especially of RTSP, we find that it is a rapid growth area - an estimated 5000 Internet Radio stations now flourish.

However, many users do not have access capacity to receiver audio (or video) at the data rates being sent. One solution is to send data at multiple rates, however this congests links near the data soruces. A distributed approach is to run transcoders that down-rate the audio (or video) to more appropriate rates. A distributed algorithm for this was simulated at UCL, and involves a TCP like feedback loop to determin inter-source and inter-transcoder data rates. The scheme also chooses near optimal placement for a transcoder w.r.t the network topology and capacity. Clearly, this is another excellent match for the ALAN technology, since it requires proxies that are located and run on demand, depending on the (dynamically, possibly changing) set of users and the network paths to the receivers.

7 Results

7.1 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.

7.1.1 Streaming Throughput

We have previously reported results for our audio transcoder [6]. These are reiterated here. The results were obtained using a Sun Ultra 1 machine running Solaris 2.5.1 as the DPS.

The main computational concern is that an audio sample can be coded or decoded in less time than the sample represents. 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.

Our transcoder also permits redundant audio samples to be sent to improve resilience to lost packets. The worst case is where 20ms ADPCM audio samples are created with two levels of audio redundancy. 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.

7.1.2 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 text 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 decompressor proxylet which decompresses it and delivers text 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

Table 1: Text Retrieval Times

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 2: Compression Times

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
NT Workstation	166MMX Intel	JDK1.1.6	14388
Linux 2.0.33	166MMX Intel	JDK_1.1.5_v5	17893

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 these 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. While we have not investigated this overhead in detail, we believe it has the following components: RMI communication to initiate proxylet loading, time to download the proxylet(s), start-up of a Java Virtual Machine for each proxylet, and creation of the various HTTP/TCP connections. We have likewise not yet attempted to optimise any of these components.

For the large file data compression and transmission are pipelined. However for tests performed locally on a switched 10Mb/s Ethernet 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 (as expected the decompression time was an order of magnitude less), which gives a compression throughput of around 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 uncompressed 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.

Table 3: Proxylet metrics

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

7.2 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.

7.2.1 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.

7.3 RLC and SOT

TBA

8 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 [12]. However this work still has 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.

Other work has demonstrated the benefits of adding services transparent to clients and servers [2, 3]. Our work also permits dynamic deployment of services.

The ALAN approach has similarities with the Active Services [1] 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 high/low bandwidth networks.

9 Conclusions and Future Work

We have proposed a novel architecture and mechanisms for the achievement of Application Level Active Networks. We have developed and tested an initial implementation of some central aspects of the overall system, namely the Dynamic Proxy Server and downloading of proxylets. The feasibility of our proposals has been demonstrated via experimental implementation of a range of proxylets. These indicate that useful services can be downloaded dynamically. Our conclusion is that the proposed architecture shows promise as an infrastructure for future active service development and deployment. It can also be thought of as an efficient and dynamic approach to the deployment of Virtual Private Networks.

However our work to date has a number of shortcomings when confronted with scenarios of wide-scale deployment. These are the subject of ongoing work. We need to develop a scalable architecture that enables dynamic DPS location. We also need to address issues of DPS security and resource management, in particular the fair management of multiple proxylet flows. Also, the concept of dynamic protocol stacks requires substantial further work.

Finally, we need a routing architecture for ALAN. DPSs will run within an ISP at many nodes. Several of the applicaitons outlined in this document would require choice of which of many DPSs should be used to provide a service, based on some metrics, and subject to some constraints. It is an opportunity to take on board some of the more recent advances in dnamic distributed multi-metric routing, such as the NIMROD work, and QingMing Ma's work on QoS routing at Cisco. Clearly the network level routing will provide connectivity, but in a full scale solution to ALAN, we need to consider application level policies (access control and transit rules and so on). The DPS routing system will have to incorporate all the feastures of a full-fledged routing system, with its own naming, addressing, neighbour and topology discovery and route planning algorithms.

10 Acknowledgements

This ALAN work is funded by British Telecom Laboratories, and resulted partly from their involvement in the industrial review for the HIPPARCH project. The additional work on adding Java versions of RLC and SOT to the system as proxylets was funded under the HIPAPRCH 6 motnh extension in 1999.

Our thanks go to Ian Marshall for the many fruitful discussions and suggestions. We thank Glen MacLarty for writing the webcache proxylet.

Our thanks to Lorenzo Vicisano and Isidor Kouvelas at Cisco Systems for guidenance on developing the RLC and SOT components.

References

[1] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 178–189, Oct. 1998.

[2] S. Ardon, R. de Silva, R. Landfeldt, and A. Seneviratne. Total management of transmissions for the end user: a framework for user control of application behaviour. In *Proceedings of HIPPARCH'98 Workshop*, pages 189–205, London, June 1998.

[3] R. Barrett and P. Maglio. Intermediaries: New places for producing and manipulating web content. *Computer Networks and ISDN Systems*, 30:509–518, 1998.

[4] T. Berners-Lee. The world-wide web initiative. In *Proceedings of the International Networking Conference (INET)*, San Francisco California, Aug. 1993.

[5] S. Casner and S. Deering. First ietf internet audiocast. *ACM Computer Communication Review*, 22:92–97, 1992.

[6] A. Ghosh and M. Fry. Javaradio: an application level active network. In *Third International Workshop on High Performance Protocol(HIPPARCH '97*, London, June 1997.

[7] M. Handley and V. Jacobson. SDP: Session description protocol. Technical report, Internet Engineering Task Force, 1998.

[8] V. Hardman, A. Sasse, and I. Kouvelas. Successful multi-party audio communication over the internet. *Communications of the ACM*, 1997.

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

[10] D. Maltz and P. Bhagwat. Improving http caching proxy performance In *Proceedings of HIPPARCH'98 Workshop*, pages 98–103, June 1998.

[11] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, Apr. 1996.

[12] D. Wetherall, J. Guttag, and D. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH'98, San Francisco*, San Francisco, Apr. 1998.