Enabling Active Networks Services on A Gigabit Routing Switch

<To be appeared on the 2nd Workshop on Active Middleware Services>

Phil Wang, Robert Jaeger, Robert Duncan, Tal Lavian and Franco Travostino
Technology Center, Nortel Networks Corp.

Key words: Active Networks, programmable networking, service deployment, Accelar, ORE, ANTS, gigabit routing switch

Abstract: Current Active Networks (AN) research projects are mainly realized in software-based network systems since available hardware lacks networking programmability. This paper studies the deployment of AN services on the Accelar Gigabit Routing Switch. The Accelar is one of the Nortel Networks programmable networking products, and uses the ASIC technology to reach the high-speed forwarding capability. The Oplet Running Environment (ORE) and the Java Forwarding (JFWD) API provide the programmable interface to the Accelar. The ORE is a pure Java environment that enables the Accelar to download and initiate network services dynamically. Using the oplet encapsulation, AN execution environments (EEs) can be deployed on the Accelar as ORE services. The JFWD API provides access to underlying hardware resources to perform network operations such as diverting packets and altering packet processing.

We demonstrate the deployment of Active Networks EEs as network services managed by the ORE, specifically, the MIT ANTS EE. We have wrapped the MIT ANTS implementation with the ORE-mandated structure and successfully run ANTS applications over a network comprised pure and ORE encapsulated ANTS EEs. In conclusion, we present observations about the AN service deployment on the Accelar.

1. INTRODUCTION

The trend in commercial-grade routers and switches is to implement ever more functionality of network in hardware; hardware implementations in turn bring ever faster processing ability, but at a loss of service customization on the data path. As more of the virtual machine is *frozen* in silicon, less are the opportunities to introduce new services inside the network.

In contrast, the *Active Networks* (AN) technology exposes a "programmable" user-networking approach that allows network services to be introduced "on-the-

fly". AN supports per-flow customization of the service and the flow is user-defined by the programmable interface. Such a customization enables Internet service providers (ISP) and even individual users to facilitate new services such as QoS (quality of Services), dynamic composite protocols, rapid network service deployment, robust network security and flexible network management.

In order to enable the AN services, network nodes must have, in addition to fast performance, the programmability with open networking APIs (Application Programming Interfaces). Traditional network nodes (e.g. routers on the Internet) enable end-system connectivity and sharing of network resources by supporting a static and well-defined set of protocols. These nodes are closed systems that allow users to perform configuration of existing services but do not allow users to add services. This limitation makes traditional nodes unsuitable for hosting the deployment of AN services.

This paper studies the deployment of AN services on the Nortel Networks *Accelar* routing-switch. The Accelar family of switches supports gigabit L3 routing, switching, and hardware filtering through the use of ASIC technology. Moreover, the Accelar supports the *ORE* (Oplet Runtime Environment [1]) and the *JFWD* (Java Forwarding) API, which provide networking programmability to the user for deploying new network services. The ORE provides an open, platform-neutral, pure Java runtime environment that enables users to download and initiate network services dynamically. Network services, including AN services, are encapsulated by *oplets* using the ORE interfaces, and injected into the Accelar by having the ORE download their oplets and execute their codes. They further use JFWD to access the Accelar hardware instrumentation.

To demonstrate the active networking capability of the Accelar, we deployed the MIT ANTS execution environment (EE) on this device. We created our ORE-based ANTS by encapsulating the MIT ANTS EE. The ORE ANTS service, called the AntsNodeService, is an ORE service that creates an instance of the unchanged MIT ANTS EE. We constructed an experimental active network with pure MIT ANTS nodes and ORE ANTS nodes (i.e., the Accelar switch) and run active applications over this network to the interoperability of our code with the "off-the-shelf" ANTS.

2. THE DARPA ACTIVE NETWORKS

The DARPA Active Networks (AN) architecture supports multiple active user interfaces called *Execution Environments* (EEs) in order to flexibly compose new protocols or deploy new services. These EEs utilize the system resources provided by the node operating systems (called the Node OS).

Moreover, AN has two key terms "*active packet*" and "*active node*". An active packet, or a capsule, carries not only a protocol header and a payload (as a regular IP packet does) but also a program code segment. The program code is executed at active (network) nodes. An active node is equipped with or can dynamically download EEs to run the code carried by active packets.

The AN research projects have made significant advancements including the MIT *ANTS* (Active Node Transfer System [6]) toolkit, The UPenn *Switchware* architecture [5], the Columbia University *Netscript* language [7], the USC/ISI *Abone*

(Active Backbone) [9] and the Active Networks protocol *ANEP* [8]. To date, they have only been realized in software that offers flexibility at the expense of performance. Furthermore, the AN goal is to bring new services to the network by injecting these technologies to real network nodes where they also gain performance by using hardware acceleration.

3. ACCELAR, ORE AND SERVICES

In this section presents an overview of the Accelar, the ORE and available ORE services so as to have a basic understanding of the Accelar programmability.

3.1 Accelar Routing Switch

The Accelar is a new Nortel product family of L3 Routing Switches and employs a distributed ASIC (Application Specific Integrated Circuit) -based forwarding architecture that can do packet forwarding at 5.6-256 gbps. The hardware is controlled using the VxWorks real-time OS, the networking programmability is provided by the ORE and related service APIs.

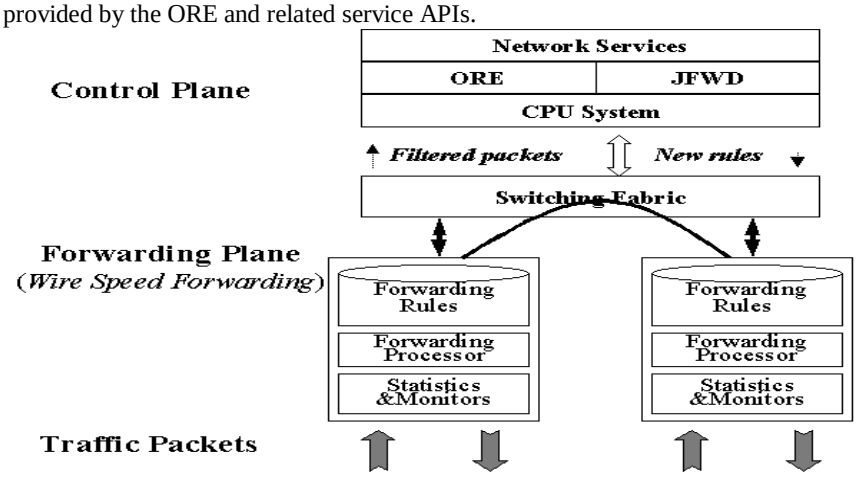

Figure 1 Accelar Programmable Networking Architecture

Legacy network devices involve the CPU in forwarding actions as well as in the control of forwarding policies. The two tasks compete the CPU power, memory and other resources, resulting in reducing the level of performance that they can achieve.

The Accelar achieves a significantly higher level of performance by separating the two tasks into control and forwarding plane operations, as depicted in Figure 1. The "forwarding" plane is implemented using ASICs that forward packets at a wire speed without consuming any CPU resources.

The CPU system is utilized by the "control" plane to process policy control such as routing algorithms as well as supports the ORE services. The ORE described in Section 3.2 is an open network-programming environment used for deploying

customer network services on the Accelar and other open platforms. Network services are managed by the ORE, and access the forwarding plane by invoking functions in the JFWD API. Through JFWD, services can alter the packet processing through the hardware instrumentation (MIBs). Some of the functions include setting packet filters to drop, modify, or divert matched packets. Filters can modify the DiffServ byte, alter VLAN priority, divert packets to the CPU, or copy packets to another port or to the CPU.

3.2 The ORE

The Oplet Runtime Environment (ORE) supports injecting customized software, including the AN EEs, into network devices. It is actually a platform for secure downloading, installation, and safe execution of Java code within a JVM (Java Virtual Machine). Downloaded code that implements a specific functionality is called a service. A service may depend on other services and offer public interfaces for other services to use.

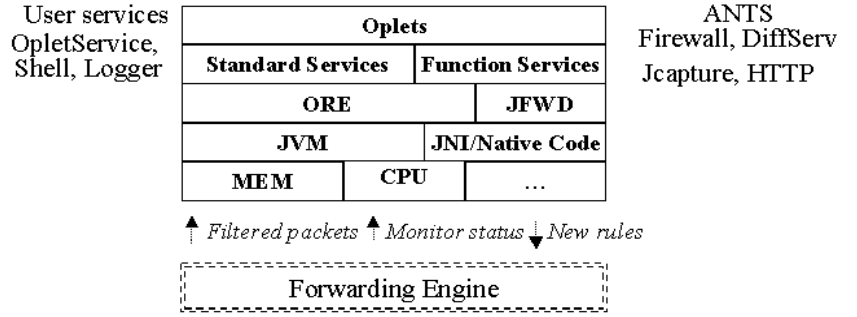

Figure 2 ORE service architecture

In order to securely download, we define the *oplet* as a self-contained downloadable unit that embodies a non-empty set of services. Along with the service code, an oplet specifies service attributes, authentication information, and resource requirements. Note that an oplet can encapsulate a service even it depends on some other services; in this case, an oplet also contains service dependency information.

The ORE architecture depicted in Figure 2 consists of the Oplet Runtime Environment (ORE), oplets, and services. Services are encapsulated by one or multiple oplets, and oplets publish those services they provide to the ORE.

The ORE provides the mechanisms to download oplets, resolve service dependencies, manage the oplet lifecycle, and maintain a registry of active services. On the network nodes, the ORE manages to install, activate, update, deactivate, kill, and uninstall oplets. Users can specify oplets that are automatically installed and activated when the ORE is started. One oplet can use a number of other services that are specified as the oplet attribute.

Prior to activating an oplet, the ORE activates all services upon which it depends. When the oplet is activated it can then register the services it provides with the ORE. The ORE tracks these dependencies between oplets and services, so that if a service becomes unavailable the oplets that are dependent on it are notified.

3.3 ORE services and JFWD

The ORE services can be classified into three categories: standard, system and customized. In Figure 1, "JFWD" is the system service for underlying packet forwarding and processing, and "Standard Services" are the ORE-specific APIs for customer service encapsulation and management. The customized services include both "Function Services" and "Oplets". "Function Services" are some common functional services that come with the ORE release, whereas "Oplets" stand for the encapsulated user-defined network services.

3.3.1 JFWD: Java Forwarding API

The *JFWD* (or Java Forwarding) API is a low-level system service, and provides platform-independent Java APIs that customer services use to control the forwarding behaviour by accessing the underlying hardware instrumentation on various platforms. It includes fundamental network mappings such as MAC address, ARP, IP routing and filters, and VLAN (Virtual LAN).

A major use of the JFWD API is to instruct the forwarding engine to alter packet processing through the installation of IP filters. On Accelar switches, IP filters are set as hardware filters that execute "actions" specified by a filter policy. A filter is based on the MAC, IP or transport protocol header, or their combination. The policy can define where the matched packets are delivered and can also be used to alter the packet content. Packet delivery options include discarding matched packets, or conversely forwarding matched packets if the default behavior was to drop them, and diverting matched packets to the CPU system (i.e., the control plane). Diverting packets allows customer services such as AN EEs to process packets. Additionally, packets can be "carbon copied" to the control plane or to a mirrored interface. The filter policy can also cause packet header contents to be selectively altered (e.g., the DSCP bits on matched packets can be set).

JFWD implementations on different platforms (e.g., Accelar and Linux) require use of native codes or communications. On the Accelar, the JFWD native codes turn out to be a wrapper around the hardware instrumentation interface.

3.3.2 ORE standard services

These standard services provide ORE-specific APIs such as service encapsulation and features such as service startup. They are essential to encapsulate customer services and conduct user interaction with the ORE. We will discuss them with the service deployment in next section.

- *OpletService* (ore.jar): the Oplet service API, extended to define service descriptions and interfaces
- *ManifestOplet* (ore.jar): the Oplet encapsulation abstract interface, implemented to create service-specific oplets
- *Start* (start.jar): ORE startup service, auto-starts specified services when the ORE starts
- *Shell* (shell.jar): telnet-like user interface service, provides shell commands to manipulate oplets and/or network services (e.g., start and stop)
- *Logger* (logger.jar): ORE log service, provides printout during running services

3.3.3 ORE function services

The ORE function services provide some common function APIs that can be used to program high-level application services conveniently. Some of them (e.g., HTTP) are platform-neutral, and the others rely on particular system supports (e.g., JPCAP on Linux).

- *HTTP* (ahttp.jar): HTTP
- *Jcapture* (jcapture.jar): Packet capturing, *sets* IP filters to divert packets to CPU
- *IpPacket(util_packet.jar)*: IP packet utility, constructs IP/TCP/UDP header and payload
- *JMIB (jmib_*.jar)*: platform MIB access, provides access to hardware instrumentation
- *JPCAP (jpcap.jar)*: packet capturing, provides use of local Berkeley *libpcap*

4. SERVICE DEPLOYMENT

Network services are programmed using standard Java classes. Before they can become the ORE services they must be encapsulated using the ORE interfaces. This section describes how a service is encapsulated and downloaded to the Accelar.

4.1 Two Oplet APIs

The ORE provides two oplet APIs for service creation and encapsulation.

4.1.1 Base service

The first API "OpletService" is a base service class that a new service extends to define its interface. That interface class includes the service description and the service function interfaces, see Figure 4 for example.

A service also has an object to implement its interface. That object class (e.g., HelloImpl.java) realizes the service function(s) as well as two private methods that the services oplet use to start and stop the service function(s). It also can import Java codes of other services or user's programs.

4.1.2 Service Encapsulation

The second "ManifestOplet" is an abstract interface that an oplet (e.g., HelloOplet) implements to encapsulate the service and register it to the ORE. ManifestOplet has two methods: startService() and stopService(), which are used by the ORE to start or stop a service.

```
                                        com.nortelnetworks.ore.service.logger.LoggerService
Oplet:          examples.hello.HelloOplet   Oplet-DocUrl:          http://www.openetlab.org
Oplet-Services: examples.hello.Hello        Oplet-ContactAddress: support@www.openetlab.org
Oplet-Description:  an example service Hello
Oplet-Dependencies:
```

Figure 3 The manifest header of service Hello: HelloOplet.mf

While loading the oplet, the ORE extracts the service information from a manifest header file like "HelloOplet.mf", e.g., the oplet name, service package and description and dependent services. This file is consistent with the service oplet.

4.2 Service Package

The Java code of a service is packed to a jar file for ORE downloading, which includes the above Java classes and other user-defined classes.

- *Hello.class*: the service interface class, extends OpletService
- *HelloImpl.class*: the service implementation class, implements the interface Hello
- *HelloOplet.class*: the Oplet class, implements Manifest and encapsulate service Hello
- *HelloOplet.mf*: the service manifest file, provides the service info

The jar file can be stored in the local ORE directory "<*OREROOT*>/ore/jars/", or uploaded to a "trusted" server for later downloading.

4.3 Service Start

There are at least two ways to start customized services inside the ORE. One way is to use the startup service (i.e., *start.jar*) that starts those services at the ORE startup. On a local host, the ORE locates the startup service using an environmental variable "*ORESTART= file:${OREROOT}/jars/start.jar*" (*OREROOT* is the ORE root directory), and also reads a property file "*start.properties*" to get the jar file names or URLs of those services.

The other is to use the ORE shell to start those services manually. Using "telnet *OREHOST* 1999" (*OREHOST* is the ORE host), one can instruct the ORE to install, uninstall, start and stop services. You may also start services using the ORE APIs.

It's noted that, if a service is dependent on other services, the ORE must download their codes and activates them before starting this service.

4.4 Service Injection on Accelar

On Accelar, both of the above ways can be used to start new services. If you use the first way, since the variable *ORESTART* cannot be preset on the Accelar, you have to input the "ORESTART" value using "telnet *Accelar* 1966".

The encapsulated service packages are stored in an external downloading server rather than in the Accelar box. When the Accelar starts, the ORE downloads the service packages using URLs and activates the services if they are specified at startup. Or, a network administrator can load those service packages into the ORE and activates them using the manual start way.

Once the activation is done, those services have been injected and become native services on the Accelar. If a service includes the appropriate APIs such as JFWD, it can access the hardware instrumentation and conduct designated network operations.

5. THE ORE ANTS SERVICE ON THE ACCELAR

The MIT ANTS [6] is a typical AN project for building and deploying new network protocols dynamically at routers and end systems. It uses mobile code, demand loading, and caching techniques, and provides a software package that comes with a toolkit and several active applications such as ping and multicast.

For the first time, a commercial-grade routing-switch has successfully hosted an Active Networks execution environment (EE) and interactive with active nodes. We built an ORE service called "AntsNodeService", which provides ANTS EE capabilities using the MIT ANTS distribution (version 1.2). Without modifying the ANTS code, the ORE ANTS implementation runs on the Accelar 1100B routing switch.

Those software packages including the ORE 0.3.3 and the whole ORE ANTS are available via "http://www.openetlab.org/downloads/", more details about deploying this service on the Accelar are stated in [11].

5.1 AntsNodeService And The ORE ANTS

The "AntsNodeService" is the ORE wrapper around the ANTS code. The ORE ANTS service is comprised of the AntsNodeService wrapper and the ANTS EE. The ANTS code is unchanged in the Java package named "ants". The AntsNodeService wrapper code is named package "com.nortelnetworks.ore.service.ants", and includes the following files.

- *AntsNodeService.java*: the AntsNodeService public interface
- AntsNodeServiceImpl.java: the AntsNodeService implementation, wraps "package ants"
- AntsNodeOplet.java: the Oplet, provides the "AntsNodeService" service
- Ants.mf: the service manifest, provide the service information

The class "AntsNodeService.java" in Figure 4 includes the ORE ANTS service description, and two methods: getNode() that connects the ANTS code, and getConfiguration() that reads the ANTS configurations to set up the service.

```
// AntsNodeService.java
/** The Active Networking Transport Protocol (ANTS) interface for the ORE environment. */
package com.nortelnetworks.ore.service.ants;

import com.nortelnetworks.ore.*;
import ants.*;  // the MIT ANTS package

public interface AntsNodeService extends OpletService {    /* service description */
    ServiceDescription DEFAULT = new ServiceDescription
("com.nortelnetworks.ore.service.ants.AntsNodeService");

    /** Gets the current Node of this instance of the ANTS service.    */
    public Node getNode();

    /** Returns a stream containing the config info. */
    InputStream getConfiguration() throws IOException;
}
```

Figure 4 The "AntsNodeService" service interface

5.2 Service Installation

The ORE ANTS service is packed using a jar file "ore-ants.jar" that includes both the MIT ANTS "package ants" and the AntsNodeService "package com.nortelnetworks.ore.service.ants". Then, it's stored in a Linux HTTP server.

We edit the ORE startup file "jars/start.properties" to add "ore-ants.jar" so that the ORE ANTS service is automatically started at the ORE startup. We also edit the ANTS configuration files "ants.config" and "ping.routes" to use the Accelar as the active node and specify the source and destination hosts. Both of them are stored in the same HTTP server too.

5.3 A Simple Test: APing

We test the ORE ANTS service simply using the ANTS Ping (APing) application. The testbed depicted in Figure 5 is an experimental net that is connected within the Nortel intranet and has four major computer boxes. This net includes both active and non-active nodes.

- *Accelar 1100 B switch*: the active router, runs the ORE ANTS
- *Solaris workstations* 1: destination active node, runs MIT ANTS
- *Solaris workstations* 2: source active node, runs MIT ANTS (and APing)
- *Linux PC*: an HTTP server, provides the ORE service jar files and the ORE ANTS configuration

When booting the Accelar (using the ORE boot image), we use "telnet *10.120.101.102* 1966" to input the URL of the ORE startup service "http://134.177.116.106/jars/start.jar". The ORE on the Accelar then automatically downloads the ORE ANTS and other services jar files from the HTTP server. When the ORE ANTS service is loaded, it downloads the configuration files and enables the real ANTS service.

We also install the MIT ANTS package onto the two hosts, and configure the source host to run the APing application sending capsules and the destination host to run the ANTS daemon receiving the capsules. On the source node, APing pops up a GUI window where we set the input fields "Num of Iterations (10)" and "Ping Interval (1ms)". Then, click the "Ping" button to start.

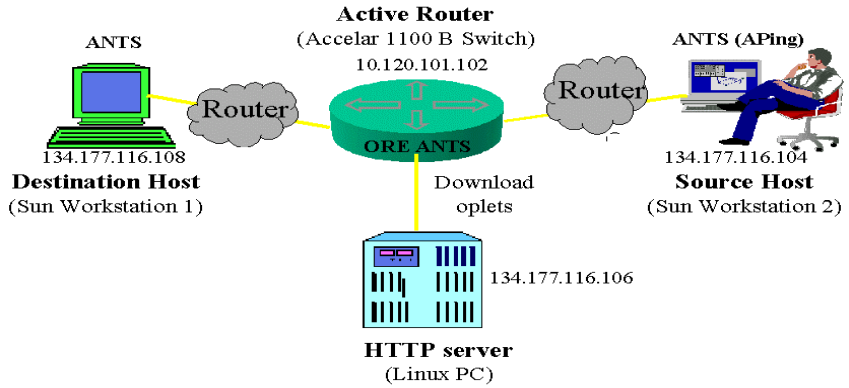

Figure 5 The experimental net of the ORE ANTS service

The source host sends out the ANTS ping capsules to the Accelar. Initially, the Accelar does not know how to process the first capsule and requests the APing application from the source of the ANTS ping capsule. The source forwards the APing code to the ORE ANTS service that installs the code in the Accelar ANTS EE. Once installed, the APing application on the Accelar "executes" the program code of those ping capsules resulting in forwarding them to the next ANTS-enabled node. When the destination sends the ANTS ping capsules back through the Accelar (now with the ANTS EE service installed), the capsules are processed and forwarded to the source ANTS EE. The receipt of all the ANTS ping capsules at the source verified that we had successfully implemented an interoperable ANTS EE implementation on a commercial-grade gigabit routing-switch.

6. ENDING REMARKS

In this paper, we mainly discuss the Accelar's programmability and use it to deploy AN services. The deployment of the ORE ANTS service on the Accelar is rather easy because the MIT ANTS software is pure Java. On the other hand, since the ORE is platform-neutral, the ORE ANTS service can be injected to non-Accelar network nodes (e.g., the end hosts in Figure 5) those run the ORE for secure downloading capabilities and management of system resources.

Like ANTS, much of the AN software is developed using Java and can easily be ported to the ORE environment (on the Accelar). Some AN research projects, such as Netscript [7], also use native codes to obtain local resource supports (e.g., routing tables and forwarding policies). When porting them to the Accelar using the ORE, they can use the JFWD API and/or other services in place of native codes.

We have shown that AN services can be run on commercial-grade hardware and benefit from the high-speed forwarding performance that the Accelar provides. They also require strong computing capability in the control plane to execute the Java codes and perform protocol algorithms. Currently, the Accelar forwarding plane is controlled by customer services through the hardware instrumentation. In the future, the Accelar will provide the ability to reprogram the ASIC so that AN and other services can re-define the forwarding engine behaviour.

7. REFERENCES

[1] R. Duncan, "The Oplet Runtime Environment", http://www.openetlab.org/ore.htm, March 2000

[2] R. Jaeger, R. Duncan, F. Travostino, T. Lavian, J. Hollingsworth, "Dynamic Classification in Silicon-Based Forwarding Engine Environments," Usenix: Intelligence at the Network Edge, San Francisco, March 2000.

[3] R. Jaeger, R. Duncan, F. Travostino, T. Lavian and J. Hollingsworth, "An Active Network Services Atchitecture for Routers with Silicon-Based Forwarding Engines", LANMAN'99: 10th IEEE Workshop on Local and Metropolitan Area Networks,Sydney, Australia, November 1999

[4] T. Lavian, R. Jaeger, J. Hollingsworth, "Open Programmable Architecture for Java-enable Network Devices", Stanford Hot Interconnects, August 1999.

[5] D. Scott Alexander, et al , "The SwitchWare Active Network Architecture", IEEE Network Special Issue on Active and Controllable Networks, vol. 12 no. 3, July 1998

[6] David J. Wetherall, John Guttag, and David L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE OPENARCH'98, San Francisco, CA, April 1998.

[7] Y. Yemini and S. da Silva. "Towards Programmable Networks", IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy, October 1996.

[8] D. Scott Alexander, et al, "ANEP: Active Network Encapsulation Protocol", Active Networks Group, Request for Comments, http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt

[9] Active Network Backbone (ABone), http://www.isi.edu/abone/

[10] Phil Wang, "Accelar and Active Networks — a white paper", http://www.openetlab.org/docs/AN_whitepaper.htm, June 2000

[11] Phil Wang, "ORE ANTS service on Accelar', http://www.openetlab.org/downloads/HOWTO.ore-ants, May 2000