Open Extensible Network Control

Herbert Bos

email: hjb1005@cl.cam.ac.uk, phone: +44-1223-334650, fax: +44-1223-334678

University of Cambridge, Computer Laboratory, Cambridge, CB2 3DQ, United Kingdom

October 16, 1999

Abstract

Network control is decomposed in six parts: switch control, resource partitioning, virtual network building, virtual network control, generic services and data-path components. Each of these parts can benefit from support for dynamically loadable code, which allows users to extend and customise the basic functionality. This is related to active networks, except that dynamic code exercises control at the granularity of connections (flows), rather than individual packets and all aspects of network control are explicitly considered. Network resources are recursively partitionable, so that dynamic code is able to control partitions of virtual networks in any way it sees fit. Policing these partitions may occur at varying levels of 'strictness'.

Keywords—**Active networks, elastic control, open control, virtual networks, policing**

1 Introduction

Recent years have seen a flurry of papers on new directions in network control, either by opening up network control [1, 2], or by allowing packets on the data-path to program the network nodes, as proposed by the active networks community [3, 4]. Drawing from experience in both of these areas, we designed and implemented a framework that enables clients to program all aspects of network control in a controlled fashion. This is done by dynamically injecting code into the various entities that constitute the network control system, subject to access control restraints. This will be defined as *elastic network control*, an implementation of which is called *Haboob*[1].

[1] The term *elastic* was borrowed from [5].

1.1 Contribution

The *Haboob* not only opens up the *interfaces* used to control and manage networks, but also allows extension and modification of existing components that exercise such control. These components need not necessarily reside on routing/switching nodes, as is generally the case in active networks. While the key idea in the *Haboob* is also 'activating the network' (and as such could be placed under the active networks umbrella), the location and role played by the active code is different from that in most active networks (see also Section 3). As such, the *Haboob* is an attempt to bridge the gap between active networks and open control.

Moreover, by now we consider the problem of making networks programmable by dynamically loadable code (or 'active packets') to be a well-established research area for which many solutions have been proposed. Although certain issues still need to be resolved, we are more interested in the next step: how best to exploit such programmability. For this purpose, an approach is taken at a slightly higher level than how active code is classified, demultiplexed, run by the appropriate execution environment, etc. However, it's also not our intention to discuss specific applications or services built with dynamically loadable code (an approach that was taken in [6, 7] and others). Instead, this paper presents both a framework and its implementation in which such questions are addressed as: which aspects of network control and management can be distinguished and how can they benefit from programmability by dynamically loadable code?

It should be noted that the programming of network nodes is nothing new: switch vendors do it all the time and reprogramming happens whenever network operators update switch software by loading in new images. What makes work like the *Haboob* different from these traditional approaches is that it (1) allows reprogramming at a much shorter time scale, (2) without having to take the controller out of service and (3) with access control at a very fine granularity (instead of only allowing the system administrator to update the control software, it is possible to specify the amount of customisation of network control a client is allowed to do on a per-client basis). Although the implementation is ATM-based (combined with IP over ATM), the issues may be relevant to any network technology with end-to-end resource reservation.

1.2 Overview

In Section 2, we clarify the terminology and overall view of network control and management maintained in this paper. In Section 3, the execution environment for dynamically loadable code is discussed. Sections 4-6 discuss how such code is loaded into the heart of various components that make up network control and management. In Section 4, it will also be explained how resource partitioning and differentiation of such policies as *policing* and *interoperability* have been implemented in a specific control system. We will show how more traditional active networks are implemented in the *Haboob* in Section 7. As an application of dynamically loadable code a continuous media video server was built. This is discussed in Section 8. Some performance figures are given in Section 9. Related work is mentioned in Section 10. Finally, in Section 11, we summarise and draw conclusions.

Henceforth, the term *active networks* as used in this paper, refers to such approaches as advocated in either the *ANTS* [3], or the *SwitchWare* [4] projects, i.e. where all or some packets on the data-path carry programs specifying how to deal with these or subsequent data packets. Using the term in this narrow sense, allows us to contrast it with the *Haboob* more easily.

2 Terminology and model

We define *control architecture* as the set of protocols, policies and algorithms used to control a (partition of a) network. Traditionally, telecommunication networks' control architectures have been monolithic and closed, but recent research aimed at opening up control and management has achieved promising results [1], allowing, for example, multiple independent network operators to be present in the same physical network and a speedup of the time in which new services can be developed and deployed in the network. The goals of this approach are what we will call *open network control*, i.e. network control based on public interfaces that can be used by any party to implement new services, independent of vendor-specific solutions. We argue that most current implementations offer only limited openness, namely at the interface level. This paper shows that it is useful also to allow dynamic extensions to these interfaces, and customisation of the components that implement them. Only then all aspects of control and management are truly open.

2.1 Decomposition of control

We identify six more or less orthogonal levels related to open network control. These levels have been implemented as separate components, allowing for independent development.

The first level in the model considers the *switch control interface*, the interface between control software and physical switch. In traditional telecommunication systems this interface is closed, but it was shown in the *Tempest* [1] how an open, generic switch control interface called *Ariel* allows for non-proprietary network control systems, i.e. that work with any switch that implements the interface. *Ariel* is sufficiently simple to be implementable for even the simplest switches. Similar interfaces have been proposed elsewhere, e.g. by the IEEE P1520 Standards Initiative [8].

The second level comprises the partitioning of network resources by a process called the *divider*.

The divider is capable of making hard partitions on a switch where each of the partitions may be controlled by a different control architecture. These partitions, known as *switchlets*, all have their own VCI space on switchports and other resources as well. In fact, to the software that is controlling the switchlet, it seems as if it is controlling its own (albeit smaller) switch, because the divider exports an *Ariel* interface for each of the switchlets. Switchlets allow a number of different control architectures to be active simultaneously on the same physical switch.

At the third level we find the *netbuilder*, the process responsible for combining switchlets to create virtual networks. Again, such virtual networks are hard partitions of the network resources and each virtual network looks exactly like a somewhat smaller physical network.

The fourth level, considers the *control architecture.* This is the software that sets up and tears down connections, and in short, actually *controls* the virtual network. Figure 1 shows three control architectures that are simultaneously active on two switches. One of them controls two switchlets on two switches. Whatever happens in these switchlets will never interfere with the other two switchlets: the operations of each control architecture are limited to its own partition of the resources.

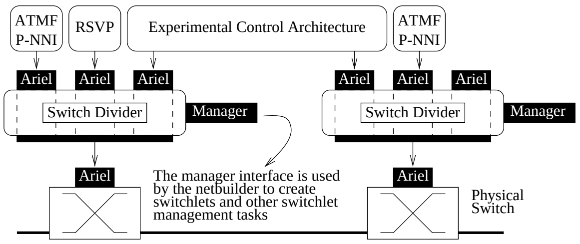

Figure 1: Multiple control architectures in a single physical network, partitioned by dividers

The general sequence of events in our implementation of this model is as follows. At start of day, the control architecture requests a virtual network from the netbuilder. The netbuilder in turn requests the corresponding switch dividers to create the appropriate switchlets on the switches, making sure that the virtual network is sensible (e.g. that the VPI/VCI ranges on neighbouring switchlets overlap, etc.). The dividers create the switchlets and return switchlet *Ariel* interface references to the netbuilder, which returns them to the control architecture. The control architecture subsequently makes calls directly over the switchlets' *Ariel* interfaces to set up and tear down connections, etc. Finally, the control architecture makes a call to the netbuilder to destroy the virtual network.

The fifth level in our model concerns *generic services*: processes providing functionality that may be used by a wide variety of clients. An example of generic services is a federated trader. A trader is used by servers to export offers for their services, which can subsequently be imported and bound to by clients. Another example of generic servers is the traffic server, discussed in Section 4.1.5, which provides applications flexible access to measurements taken from a switch.

The sixth and last level comprises *data-path components*, i.e. components that reside on the data-path and participate in network control. Examples include filters, transcoders, and other dataprocessing engines. Unlike such active networks approaches as ANTS [9] and SwitchWare [4], we treat data-path components separately from the other components. Normally, there will be no interference with the data-path at all. Only in the (exceptional) event that data processing is needed, will data-path components be instantiated. An example is given in Section 7.

We now have a complete decomposition of network control and management. The model can easily be extended if necessary in the future. The question that we ask ourselves is: how can each of these levels benefit from dynamically loadable code and what are the issues? This paper argues that it is useful to provide such dynamic customisability in all of these levels.

2.2 Lack of flexibility

Open control such as described above (and in projects such as *xbind* [10]) allows for a speedup in the development and introduction of new services compared to traditional control of communication systems. However, so far the functionality of each of the levels in the model is still fixed (specific problems resulting from this rigidness will be discussed throughout the text). This paper argues that it is useful to allow clients to extend and override the functionality of all levels of network control and management in a controlled way. Furthermore, although in principle it is possible to build and use the ideal control architecture for every application that has specific network needs, it seems unlikely in practice that there will be more than a few of these control architectures active at the same time. The reason is that rather than implementing a special-purpose control architecture, most application writers will be content with using an existing control architecture that is closest in functionality to what the application really needs. Allowing clients to customise existing network control solutions allows them to provide this extra functionality themselves.

3 Loadable code

The active networks community states that instead of standardising the computation performed on every packet, we should standardise the computational model [3]. No distinction is made between control and data. Although we agree with the statement in principle, we think that it should be extended explicitly to the control plane. As mentioned above, the *Haboob* treats processing on the data-path as the exception rather than the rule and therefore the role played by customised computation on individual data packets at every node in the network is relatively minor. Although we do allow such customised computation, as will be shown in Section 7, its applicability is considered to be rare, as most packets will probably want to be switched at the highest possible speed (i.e. without interference by application-specific computation). For the control plane, however, the inverse is true. As the execution of control code has no direct impact on the speed at which the data travels, the control plane is the ideal location to allow application-specific code to implement new services. The granularity of control by dynamic code is the connection (or flow) rather than individual packets that make up these flows (except for data-path components).

The active networks' approach is to treat *every* packet on the data-path as a program (or *capsule*) as in [3], or at least allow *some* of the packets to carry program code as in [4]. Instead, we propose not to interfere with the data path at all. Also, if the network technology allows the separation of control from data, this allows important signalling messages to get through even in the face of congestion. These messages may contain the code (in the form of DLAs) to deal with the congestion. This separation is a standard feature of ATM, but requires some work in technologies like IP (e.g. different queues for these two classes of traffic). Fortunately, quite some research is currently directed towards achieving this, e.g. IntServ [11].

Moreover, instead of simply allowing clients to push code into routers, we consider each of the tasks in Section 2. In other words, we address such questions as: how can we allow clients to extend the functionality of the switch interface, who should we allow to do this, what does it mean when we allow clients to program a control architecture and to repartition resources, how much freedom do we give them when controlling these resources (e.g. with respect to policing), how can we permit an application $\mathcal{A}$ to specify the way in which interoperability between its own control architecture and neighbouring control architectures should be handled (for $\mathcal{A}$'s flows only and not for all others), etc.

We will show that this approach is also different from what is commonly known as intelligent networks (INs) [12], in that the code can be both active and passive (e.g. waiting for a trigger). Additionally, elastic control operates at all levels of network control, rather than just the call state. Another way of saying this is that INs are easy to implement in the *Haboob*, while an inverse mapping is not possible.

3.1 The Sandbox

The elastic behaviour starts with the instantiation of an execution environment for the DLAs. This environment will be called the *Sandbox* and is implemented as a single object. The Sandbox is language and implementation-independent and only its interfaces are specified. Apart from the default language support in a Sandbox, functionality can be added in the form of modules.

A full implementation of the Sandbox comes with at least one module, which implements the simple uniform framework for interaction *(SUFI)*. The SUFI provides operations that allow DLAs in different Sandboxes to communicate, whilst preserving location transparency, either via remote evaluation [13], or via more traditional remote procedure calls (RPCs). These SUFI operations are extremely simple. For example, parameters in remote evaluations are always treated as byte sequences. The SUFI relies on *adaptors* to do the marshalling, unmarshalling and type checking of the parameters at runtime. In the case of RPC, adaptors could be generated from interfaces automatically and prepended to a remote evaluation call that encapsulates the RPC (as a remote evaluation). This is currently not done. The SUFI also provides operations to the DLAs for:

- exporting a DLA's interfaces;
- access control to a DLA's operations (currently based on capabilities);
- security: for example, it provides a method for establishing trust using a trusted third party;
- reflection: DLAs can find out information about the Sandbox and also which interfaces and operations it supports and how to use them;
- threading and semaphores for intra-DLA interaction.

It is possible to register new modules with the Sandbox, the result of which is that the commands contained in the module's interface are made available to the DLAs. The remote evaluation paradigm is quite powerful and allows clients to extend the functionality of a Sandbox. It works as follows. A client with the appropriate access rights presents a bit of code, called a *granule*, to a remote Sandbox for evaluation. Granules may be *active* or *passive*. For example, an active granule may iterate over a switch's management information base to see whether particular connections exist and send back the result. Passive granules may simply install the function for doing the iteration; the code will not be activated until it is explicitly called. DLAs can be thought of as being made up of granules. Granules are defined as fragments of code that can be autonomously evaluated.

Remote evaluation of granules, combined with the ability to destroy granules, provides a way to implement replicating and migrating DLAs. Whether or not the state is automatically sent as part of the migration is beyond the scope of the SUFI. This depends on the language environment. The current implementations require programmers to explicitly gather the state that they would like to use at the remote Sandbox. Note that granules and migration also allows for *disintegration* and *integration* of DLAs. We speak of disintegration when a DLA is decomposed in a number of granules which migrate to different sites ("spreading out" the DLA). Integration is the opposite effect.

3.2 Implementation

Sandboxes have been implemented for *Tcl*[2] and Java, while work on Python is underway. In each case, the underlying Sandbox for the language was implemented as a single C++ class, so it can be easily incorporated in existing code that was not written with elastic functionality in mind. Note that remote evaluation of Java granules by a Tcl Sandbox is bound to fail. Instead, the SUFI demands that in the case of different language environments, RPC can still be used for interaction according to a traditional client/server paradigm.

Quite some thought has been given to safety and security. Careful shielding, strict access control, trust establishment and controlled access to resources are part of the full Sandbox implementation. Controlled access to resources should also prevent DLAs from using too much CPU time, memory, etc. This can be enforced by using an operating system such as Nemesis [14], which allows one to partition resources in the operating system itself. The current implementation, however, still uses Solaris Unix, which is not capable of strict resource control. A thorough discussion of security issues is beyond the scope of this paper.

4 Elastic control architectures

There are many ways to build a control architecture, each of which may be useful in certain environments, but not in others. Another way of saying this is that if control architectures represent the boundary between client applications and the network and its primitives may be considered the API for dealing with the network, this API is tailored to a certain class of applications. Even so, we have made an attempt at a general solution for control architectures in the *Noman* class of control architectures. Noman will be discussed in Section 4.2. First we will deal with some aspects of a specific control architecture called the *Sandman*, as it demonstrates some of the power of elastic control.

4.1 Sandman

Sandman is a distributed control architecture, built on top of a CORBA compliant DPE. Figure 2 shows the components that make up the basic Sandman. An hierarchical trading mechanism allows clients to find the control architecture and other services. At start of day, the Sandman obtains a virtual network from the netbuilder in the form of interface references for switchlets. It performs control operations, such as connection setup, teardown, etc., by calls over the *Ariel* interfaces of these switchlets. The important control decisions are made by the connection manager and its support

[2] a parameter supplied to the constructor decides whether to use *Safe-Tcl* instead of standard Tcl

modules, such as call admission control (CAC), routing, etc. The switch fabric components are responsible for actual connection setup across individual switches, while the local host managers represent the local interface to the control architecture. Clients make requests to the host manager, which tries to reserve the local resources and if successful, forwards the request to the connection manager. Once a connection has been successfully established the host manager calls back the client, providing it with information about the established connection (e.g. the VPI/VCI pair in the ATM implementation). As we shall see, the Sandman has strong support for advance reservations.

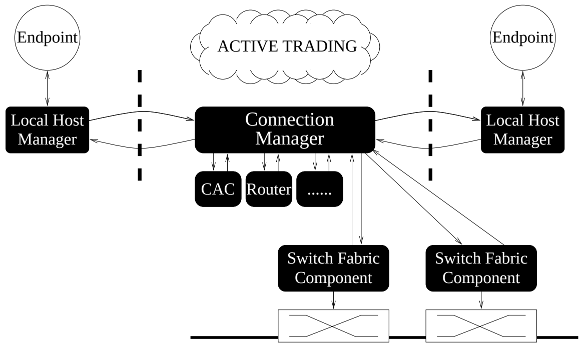

Figure 2: Components of a control architecture

4.1.1 Basic operations

In its default configuration, the Sandman offers only a few commonly used operations that are expected to be sufficient for a large class of applications:

1. *Unicast connection.*

   The simplest and probably most common operation is the connection from source to one or more sinks for a particular time interval with particular characteristics (e.g. a peak rate of 1000 cells/s). A clients submits such a request to the control architecture and if the call admission control (CAC) module accepts the request, the client is guaranteed that the connection will be set up in that time interval. Traditional, immediate, connections simply leave out the interval, in which case it defaults to $[now, \infty\rangle$.

2. *Multi-source, multi-sink connections.*

   There is also a small number of more complicated types of connections, such as a connection that is time-shared by multiple sources and which may have multiple sinks each with its own and possibly overlapping time interval. For more details, see [15].

3. *Information gathering.*

This is a rather wide-ranging class of operations to discover certain things about the state of the network, the topology, routes, available bandwidth on a certain switch port, etc.

4. *Loading application-specific code.*

   Allowing users or applications to load their own code into the Sandman allows them to exploit application-specific knowledge at a very low level. We will discuss loadable code in Section 4.1.3.

5. *Reservation of arbitrary sets of resources.*

   The reservation of arbitrary sets resources by partitioning (and repartitioning) existing sets is described in Section 4.1.4.

The first two of these operations allow for reservation in advance, so that guarantees about the availability of the required resources at some time in the future can be given. These are probably also the most commonly used operations that are expected to be sufficient for the majority of applications. We call these the *primary* operations. All other operations mentioned above are called *secondary* operations.

4.1.2 Call admission control

Admission control in the Sandman is based on the conviction that it is prohibitively difficult to accurately characterise the behaviour of unknown sources. This means that statistical multiplexing cannot be derived from careful analysis of the supplied traffic descriptors. Instead, we base our CAC decision on a new connection's supplied *peak rate* only. To alleviate this potentially over-conservative reservation mechanism, we extend it with measurement-based effective bandwidth estimation, based on [16]. For this purpose, we introduce *traffic servers.*

Traffic servers take continuous measurements of the traffic on the ports of a switch and compute the effective bandwidth of connections belonging to the Sandman's virtual network. This effective bandwidth is subsequently used for CAC decisions. In other words, for *unknown* sources, CAC is based on peak rate alone, but once connections become active an estimation is made of how much bandwidth they really need and this is used for any further CAC decisions. Traffic servers may vary from switch to switch and can be plugged in on the fly. It is conceivable that traffic servers are plugged in for certain switches and indeed certain switch ports, but not for others (in which case the default conservative CAC is used). A more detailed explanation of the Sandman's CAC can be found in [15].

4.1.3 Dynamic code loading

Dynamic code loading is supported by the instantiation of Sandboxes in the control architecture into which clients can load their DLAs. The DLAs have access to all primary and secondary operations. Observe that this does not really alter the functionality of the Sandman. DLAs simply act as normal Sandman clients that have moved into the Sandman's address space, so all it potentially gains is an improvement in performance. Next, we will demonstrate how the functionality can be easily extended using the same principle.

It is important to make a distinction between an application's *resource manipulation behaviour* and its *resource reservation behaviour*. Resource manipulation behaviour is defined by the application's actions and operations on resources under the implicit assumption that the resources are available for it to use. This type of behaviour includes allocating resources to connections, freeing a connection's resources, etc. In resource manipulation behaviour, applications generally don't worry about the availability of the resources. Resource reservation behaviour on the other hand concerns itself solely with ensuring that certain resources are available at certain times. In general, it does not care what the resources are used for (or whether they are used at all)[3].

Due to space limitations, this paper focuses on how DLAs can be loaded that specify mainly a client's resource manipulation. However, the Sandman also gives applications complete freedom in specifying their reservation behaviour, again in the form of DLAs. For example, it is possible to specify that every day from 9am to 5pm a network should be reserved with a capacity equal to $f(t)$ bps, where $f(t)$ may be as arbitrary as "the number of seconds remaining until the year 2000". Note that such DLAs give rise to issues to do with scheduling; for example because a resource reservation DLA has to be evaluated each time a new reservation is made for any of the resources associated with it. In practice therefore, one will probably want to restrict both the amount of resources (e.g. CPU time) that a DLA can use and the expressiveness of the language for this type of DLAs.

4.1.4 Recursively partitioning networks

The *primary* operations are expected to be sufficiently expressive for a large number of applications. Some applications, however, have specific needs, so in order not to restrict them, we propose to give these applications a number of resources which are theirs to use as they please (i.e. without any connections imposed on them). For this purpose, the Sandman includes the possibility to make (possibly advance) reservations for what we have called *netlets*, i.e. small virtual networks in a larger virtual

[3]Note that many of the basic operations, such as to set up end-to-end connections incorporate both reservation and manipulation behaviour

network. Netlets consist of (a share of) an arbitrary number of resources within the encompassing virtual network. The Sandman exports low-level primitives to control the individual resources of a netlet. For example, it is possible to create arbitrary connections across a single switch (assuming the resources for this connection are part of the netlet). This is quite different from traditional control primitives in communication systems, which allow clients to set up connections between *endpoints* only.

We call these low-level primitives the Sandman's *tertiary* operations. Tertiary operations are currently only accessible to DLAs. It is possible to associate a DLA with a netlet, which means that DLAs can control the netlet's resources at any level of granularity it chooses. It can implement application-specific functionality in an efficient manner (all communication takes place in the same address space rather than across the network). In this way, DLAs can also implement new functions (e.g. to create up a new *type* of connection), which it subsequently exports. From then onwards, external clients with the appropriate access rights, can call these functions, as if they are part of the normal control architecture's repertoire. This is one of the things that distinguish netlets from normal virtual networks (and from such parent-child virtual networks as discussed in *Genesis* [17]). The interfaces and DLAs are illustrated in Figure 3.

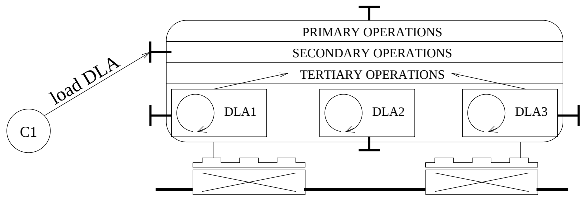

Figure 3: DLAs in the Sandman

Netlets can be created recursively. In other words, it is possible to create netlets in netlets, which enables applications to repartition network resources in an almost unrestricted manner. In fact, the encompassing virtual network of Section 2 itself is really a netlet (the so-called *level-0* netlet). Repartitioning network resources extends the idea of switchlets into the control architecture. Additionally, netlets present new opportunities for specialising policies in virtual networks. As examples, we will consider policing and interoperability.

4.1.5 Application-specific policing

Level-0 netlets must be policed, because misbehaviour in one level-0 netlet $\mathcal{N}$ should not affect connections in any of the other level-0 netlets. Given this policing, however, we can decide not to police at the netlet level (or police more loosely), for example because we *trust* the applications in the netlet to behave. The reason for this is that even if connections in a netlet misbehave, the problems will be limited to $\mathcal{N}$ only and not propagate to the outside world.

Netlets therefore allow us to differentiate the policing policy in the network. Given hard (in-band) policing for level-0 networks, we can now decide to police specific higher-level netlets only very loosely (e.g. by periodically taking measurements from switches to see if they have exceeded their allocated bandwidths) and certain other netlets not at all. In fact, the *looseness* may vary from netlet to netlet[4].

For loose policing, we modified the traffic servers of Section 4.1.2 by extending it with a Sandbox. A client of the traffic server (such as the Sandman) is able to load DLAs into this Sandbox, which determines in what sort of statistics the client is interested. For example, it is possible to load code that specifies (1) that a certain netlet should be checked every second to see if its connections stay within the capacity allocated to the netlet, and (2) that the control architecture be informed if this is not the case. This is an example of loose, or *a posteriori*, policing. Elastic traffic servers allow clients to program exactly what sort of information they receive about the measured traffic. In other words, the elasticity makes them much more general-purpose than the simple CAC tools introduced in Section 4.1.2. Such traffic servers are examples of elastic *generic services.*

4.1.6 Interoperability

Interoperability between control architectures is extremely important in an environment where multiple control architectures are expected to coexist. Existing solutions simply map the primitives of one control architecture to their nearest match in the neighbouring control architecture. For example, suppose that two endpoints A and B in neighbouring control architectures CA_A and CA_B need to communicate and that CA_A supports advance reservations and CA_B doesn't. In all probability, an *advance* reservation request from A in CA_A will then be mapped onto a *direct* reservation in CA_B, so the availability of the resources at the time when they are needed is ensured. We call this the simple hop-by-hop solution. There are two problems with hop-by-hop interoperability.

The first problem concerns the *functionality degradation* at the domain boundaries. This is a serious problem if two feature-rich control architectures communicate via a third control architecture that has a less sophisticated API. Following through the example of advance reservations, if CA_A and CA_B both support advance reservations, but some intermediate control architecture CA_C (located between CA_A and CA_B) doesn't, an advance reservation request initiated in CA_A will be translated to an immediate reservation in CA_C and will *never* be "upgraded" to an advance reservation again. In other words, the functionality *degrades* as domain boundaries are crossed. Our solution involves the establishment of

[4] In other words, netlets are light-weight virtual networks. In this sense the relation between a higher-level netlet and a virtual network is similar to that between a thread and process.

explicit inter-domain signalling channels between CA_A and CA_B using the simple hop-by-hop solution. The inter-domain signalling channel is used to carry higher-level control messages between feature-rich control architectures. The control messages are simply forwarded by CA_C (tunnelling). A complete description of inter-domain signalling channels is given in [18].

The second problem concerns the rigidness of existing interoperability solutions. For example, in the above it was assumed that an advance reservation is always translated to an immediate reservation in CA_C. This may be the right solution in some cases, but it is not necessarily the best solution in all cases. Rather than making an immediate reservation to emulate an advance reservation, pinning down resources for a very long time, an application may prefer not to prereserve in CA_C at all and simply hope for the best, i.e. only *try* to reserve the resources when they are needed and accept the consequences if this fails. This would be a very good solution if the application *knows* that bandwidth will not be a problem in CA_B in the requested time interval.

The point is that only applications know which mapping is best for them. To overcome the problem of fixed primitive mapping, the Sandman allows applications to load in their own specific mapping policies which overrule the default mapping. Observe that this is not without risks. For example, we must guard against an application overriding the default operation mapping with its own application-specific mapping (which may well be faulty), thereby affecting the interoperability of all applications. This is where netlets are useful: we restrict the application-specific mapping to a specific netlet. Other netlets or connections outside the netlet are not affected by the new policies at all. Moreover, DLAs can be employed to extend the functionality of the control architecture, by creating innovative connection types with the help of netlets. These functions can then be exported and called from external clients. It is now possible to specify the interoperability of particular new connection types in application-specific manners, which would be hard to do without netlets.

4.2 Noman control architectures

The Sandman is an example of a home-grown control architecture. It is useful in environments where such things as resource repartitioning (netlets) with DLAs, or advance reservations are required, but maybe not in many others. It is very difficult to solve the control architecture problem in a general way, without referring to a context. *Noman*, our attempt at finding a general solution, therefore, does not aim at designing the ultimate control architecture with functionality that will satisfy all. In fact, Noman offers no control architecture functionality at all, instead providing clients with the means to implement their own functionality. Noman consists of a Sandbox which is extended with a network control module, known as the *netcontrol* module. The module provides DLAs with a simple API which allows them to build, expand, control and destroy virtual networks. So on top of the Noman, a DLA can be loaded which contacts the netbuilder and creates a virtual network, consisting of a set of switchlets. The DLA obtains handles on the interface references for the switchlets that are returned as a result of this. These handles can subsequently be used to invoke *Ariel* operations on the switchlets, e.g. to set up or tear down connections. When the DLA is done with the virtual network, it calls an operation in the *netcontrol* module to remove it. In other words, Noman is not really a control architecture at all. It is merely a component that allows rapid development of new Noman-based control architectures. The dynamically loadable control architectures will be called Noman control architectures.

Using Noman, clients are able to upload their favourite control architecture functionality ("dial a control architecture") and even modify, or add to it, on the fly. This is attractive for at least two reasons. First, it is possible to generate libraries of various control architecture components (e.g. routing, connection setup and CAC components), out of which components can be picked and added to the Noman in a mix-and-match fashion, in order to create the control architecture of choice. This is related to the way components are bound together in *xbind* [2], but is slightly more general as components can be customised and replaced on the fly. Second, it allows for *vertical integration*. For example, it is possible to load the application on top of a set of Noman components to make an application manage its own resources. Application and control architecture tasks are then integrated in the same address space, resulting in considerable performance improvement. The underlying Noman components could be shared by a number of applications or they could belong exclusively to a single application, in order to prevent something similar to what in the context of operating systems is called *crosstalk* in shared components [14].

5 Elastic dividers and switch interfaces

It is the switch divider's task to partition resources on a switch. In the simplest case, this means that each partition (switchlet) obtains its own VPI/VCI space (and possibly some other simple resources as well). It interacts with the outside world via two types of interface: (1) the *management interface*, which provides operations for the creation or destruction of switchlets, and (2) *Ariel* interfaces (one per switchlet) to perform control operations. There are therefore also two places where the instantiation of Sandboxes would be useful. The first is in the management component where new switchlets are created. The second is in the *Ariel* instantiation itself, i.e. in the switchlet.

5.1 DLAs in an *Ariel* Sandbox

Placing a Sandbox in a switchlet itself allows one to load application-specific code as close as possible to the switch (this could be on the switch itself). By using a new module, known as the Sandbox *Ariel* module, this code is able to call any switch control operation available to a traditional control architecture, the only difference being that the call need not be transmitted over the network. If the divider runs on the switch, this could even be a simple function call in the same address space. Such active switchlet DLAs are called micro control achitectures. Micro control architectures are either controlled from external applications, or in a more extreme case, contain their own application code. The latter case will be termed *application-specific* micro control achitecture. Micro control achitectures that are controlled from external applications can easily extend *Ariel*'s functionality by implementing for example such things as batch connection setup or teardown. The external application simply calls the new operation once with the appropriate parameters and the micro control achitecture sets up (or tears down) the appropriate connections.

Furthermore, the *Ariel* Sandbox supports an *override* operation, which allows it to safely replace, modify, or extend the standard *Ariel* operations. Such functionality enables one to change the behaviour of a switchlet. As a trivial example, consider a DLA which overrides the connection setup and teardown operation with the empty operation. This effectively removes these operations from the switchlet's repertoire. The control architecture controlling the switchlet can still perform all other *Ariel* operations (e.g. gather port and connection statistics), but it is not allowed to create or delete connections.

A more interesting example that has been implemented, deals with VPI/VCI space virtualisation in an ATM network. Recall that switchlets constitute a partitioning of (among other things) the VPI/VCI space. This means that CA_A obtains a different VPI/VCI range than CA_B. However it is well-known that certain control architectures depend on specific VPI/VCI values for signalling. UNI signalling, for example, uses VCI 5 on VPI 0 for signalling messages [19], while both IP Switching [20] and FORE SPANS [21] depend on VPI 0 and VCI 15. Problems arise when one wants to start up a control architecture that relies on a particular VPI/VCI value V, when V was already assigned to another switchlet. This can be solved easily, using the override operation in the *Ariel* Sandbox. All operations may be overridden to map V onto one of the switchlet's own VPI/VCI values V^*. This is transparent to the control architecture, which sends the usual control messages for VPI/VCI value V. In fact, it is possible to completely virtualise the VPI/VCI space. In other words: allow the control architectures to use the entire spectrum of VPI/VCI values supported by the switch (or even more) and install a DLA at the switchlet, which maps these virtual VPI/VCI values to real VPI/VCI values in the switchlet's address space (if needed, the *Haboob* allows real VCIs to be pinned down at the edge of the network).

If control architectures span multiple switches, this means that for each of these the virtualisation DLA needs to be installed. This is a responsibility for the netbuilder (discussed in Section 6), which knows about the allocated resources of netlets and is therefore able to install automatically the appropriate mappings in the switchlets at network creation time.

5.2 DLAs in the switchlet management component

The need for elastic runtimes in the process of creating switchlets itself may be less obvious. One simple use is that it allows network administrators to install simple extensions to the default network functionality by automatic instantiation of an appropriate DLA in an *Ariel* Sandbox whenever a switchlet is created. This DLA can then extend or override the *Ariel* operations as described in the previous section. For example, the management DLA may be triggered each time a "create switchlet" operation is called (i.e. it has overridden the method to create switchlets). It can then examine the request and depending on certain conditions (e.g. the identity of the requester) it may create a new switchlet with an *Ariel* DLA which overrides the operations to create and delete connections with the empty operation. This ensures that this switchlet's control architecture is allowed to do anything on the network, except create and delete connections. A more interesting use of management DLAs implements switchlet aggregation, whereby the creation of a switchlet really creates a whole cluster of switchlets which are presented to the control architecture as a single (large switchlet). These aggregate switchlets are used to improve scalability. Due to space limitations we will not discuss aggregate switchlets any further

6 Elastic network builders

Netbuilders are responsible for combining switchlets into virtual networks (level-0 netlets). Because they are essentially shared servers with very little functionality that is specific to any particular control architecture, netbuilders are interesting when it comes to adding elasticity to them. The question arises: who decides how virtual networks should be created? Until now, this is decided once and for all by the implementers of the netbuilder. The static netbuilder has a fixed functionality—virtual networks are built by associating switchlets in a very implementation-specific manner. This paper argues that this is unnecessarily restrictive. Although it is beyond the scope of this paper to discuss the netbuilder in much detail, we briefly outline the issues and show where the use of DLAs has lead to improvements compared to implementations that only use static code.

For example, there is no reason why control architectures should not be able to push code into the netbuilder when this code is restricted to using only those operations that were exported by the netbuilder anyway. In other words, it should be possible to load active or passive DLAs into a netbuilder which, besides the SUFI, offers the exact same API as offered to control architectures. Doing this may optimise performance (especially since netbuilders may run very close to or even on a switch as well). The DLAs in turn are able to build new services by combining existing netbuilder services and exporting interface references for these services.

So far, however, only the ability to allow DLAs to (re-)combine existing netbuilder functionality has been discussed. A more interesting question is: who is allowed to add to, remove from or modify the netbuilder's default functionality? It seems reasonable to give at least the system administrators such powers. After all, they may want to override certain operations to make them include extensive logging, or to disable certain functions, or even to preclude the use of certain switches for certain operations. The system administrators then, should also be allowed to grant these privileges to other network users (in the same way as a system administrator is allowed to grant the root password of machines to certain, trusted users).

The new functionality that is thus created by the system administrators can subsequently be used by non-privileged DLAs that present the appropriate capability. In this way, the netbuilder's functionality can be changed and extended on the fly in a safe manner. In the *Haboob*, the netbuilder allows clients to load code into a shared Sandbox. The code is able to use all the basic operations offered by the netbuilder, as well as modified or extended operations implemented by the system administrator's DLAs (provided the right capability is presented).

A very simple example of a netbuilder extension is one which automatically pushes DLAs in the switchlets that are created on its behalf by the divider. In this way, we made a netbuilder push 'virtualisation DLAs' in each of the switchlets, so that the VPI/VCI spaces of these switchlets were completely virtualised (as discussed in Section 5.1). The netbuilder makes sure that each virtual VCI is mapped on the appropriate 'real' VCI (taking into account that neighbouring switchlets need the same mapping).

7 Active Networks

We recognise the value of in-band computation, as advocated by the active network community, in certain cases. For example, [7] shows as an example how a protocol "booster" can extend the functionality of a protocol on the fly, by adding forward error correction (FEC) on the datapath when it is found that one of the links is very slow and unreliable. The booster only affects that particular link. Similarly, [6] shows how an "active bridge" between two ethernet segments allows for upgrading to new versions of network software with minimal disruption.

So how do we deal with these problems in the *Haboob*? The answer is simple: if and when necessary we build a simple active network node on the fly. Consider the example of a protocol booster. Whenever the decision is made that a particular link is unreliable and needs FEC, it is simple to dispatch a DLA that reroutes the existing connection to a control entity near the link. The control entity adds the FEC and returns the code to the data path. The FEC addition can be done by a DLA running in a Sandbox in the control entity. The protocol booster mechanism is illustrated in Figure 4. Note that this does not differ in any way from the protocol boosters in [7]. In fact, we have built a very simple (off-switch) active node. This is an example of an elastic *datapath component*.

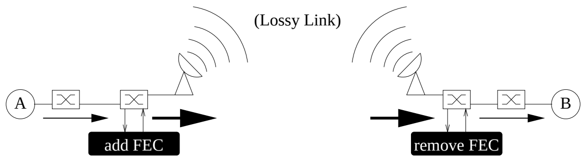

Figure 4: Active protocol booster in the *Haboob*

8 A Distributed Video Server

As a further demonstration of the *Haboob*, a distributed video server (DVS) was built on top of Sandman. The aim of the DVS was to provide a virtual big disk to low-end users that do not have enough storage space to store, for example, feature-length films on their local disks. The DVS, which is entirely implemented in dynamically loadable code, is able to use *all* disks in the network, not just the user's local disk, for storing very long video files. In recording mode, the video data is automatically striped over the various disks (using coarse-grained striping), according to a user-defined load-balancing policy. This means that the appropriate connections are set up at the appropriate times to each disk on which a particular segment should be stored. In playout mode, the DVS finds all the segments that make up the video and sets up connections from the corresponding disks at the appropriate fashion to ensure smooth playback. The sink is not aware of this. It simply receives on a single VPI/VCI pair. The sources are merged in a resource efficient manner, by reusing as much of the existing connection as possible. The DVS is described in detail in [22].

9 Performance

Because of space limitations, we are not able to discuss results in great detail. As an indication of performance we measured the connection setup and teardown times in a configuration where the endpoints of communication reside on Sun UltraSparcs and the control architecture, divider and netbuilder all run off-switch (also on UltraSparcs). As shown in Table I, a setup across a single FORE ASX-200 switch using Sandman took as little as 7.5 ms[5]. A teardown took on average 2.6 ms. This compares favourably with both ATMF UNI signalling as reported in [23] and *xbind* open control as reported in [10] (see Section 10). This should be taken as a rough comparison only, as the configuration and load of the switches in the various scenarios were almost certainly different (which has significant effect on the performance). Replacing Sandman with a Noman control architecture adds approximately 1 ms to the setup and teardown time. These figures are very good and prove that off-switch control using DLAs need not result in poor performance.

switch	**software**	**location**	**t (ms)**
Unknown	Xbind/GSMP	off-switch	16
DEC AN/2	UNI (Q.Port)	off-switch	21
ASX-200WG	UNI	on-switch	20
ASX-200BX	UNI	on-switch	10
ASX-1000	Sandman/GSMP	off-switch	15
ASX-200	Sandman/SNMP	off-switch	7.5

Table 1: Comparison of switch setup times

10 Related work

Several solutions to make networks programmable have been proposed in the last few years. We briefly discuss active networks, open control and intelligent networks. Active Networks are packet-switched networks where each packet may carry executable code. In [3] these packets are called *capsules*, i.e. little programs with embedded data that are evaluated in a transient execution environment, allowing network nodes to process the data in an application-specific way. The execution of the packets in the data path is strongly related to the speed with which these packets can travel through the network. Note that the Operation, Administration and Maintenance (OAM) cells in ATM offer a similar, albeit much more restricted functionality (the OAM cell effectively carries one of a finite number of predefined programs).

[5]The GSMP server used for the ASX-1000 was a rather buggy experimental implementation, which explains why it is so much slower than the SNMP server on the ASX200. In addition, its control processor, a i-960, is much slower than the Pentium II used for the ASX200.

The Switchware project [4] at the University of Pennsylvania, is similar to this, but does not require each packet to be a capsule (although some packets may carry code). SwitchWare is designed as a three-level architecture, where the first level comprises the active packets, the second level concerns so-called active extensions (programs that can be dynamically loaded over the network and offer functions that can be used by active packets) and the third layer is the infrastructure that enforces the rules for dynamic code loading and takes care of resource allocation. Formal methods are used to prove security properties of the dynamically loadable programs.

In [5] a solution to network management using *delegated agents* is proposed, where the agents are dynamically loadable code that can be dispatched using a so-called delegation protocol to an executing elastic (extensible) server. This helps prevent the explosion of management traffic from all over the network to a central site. Delegating management also makes the control loop (from managing code to managed device) smaller, decreasing the probability of failure at times when there are problems (and management is needed the most). The delegated agents approach has been quite influential. For example, another project that investigates the use of agents for network and systems management is INSERT [24], where Java agents are used to implement managed objects (similar to those of SNMP and CMIP) and can be instantiated dynamically.

The second approach to making networks programmable is known as *open control.* Early implementations of such control are *xbind* [10] and the *Tempest* [1] (of which the *Haboob* is a continuation). *Xbind* constitutes a framework for the creation, deployment and management of multimedia services on ATM networks with end-to-end QoS guarantees. Services are created by interconnecting (binding) abstract representations that model the local states of network resources including links and switches. For this purpose, these abstractions are stored as an organised collection of interfaces, called the Binding Interface Base (BIB).

Closely related to netlets is the more recent *Genesis* project [17] (netlets were introduced in a primitive form in [15]). Like the research presented here, Genesis enables recursive partitioning of network resources. The partitions of network nodes, called "routelets", are created by a *parent* control program, but controlled by an independent *child* control program. However, the Genesis model is much more complex than that of netlets. More serious is that the hierarchy of nested routelets seems reflected in the datapath. Each routelet consist of a myriad of components and services and part of this comprises the routelet's transport module, which resides on the datapath. The transport module includes a virtual input port, a forwarding engine (that processes the data) and a virtual output port. An incoming packet is classified, for example, as belonging to a specific (higher-level) routelet and demultiplexed to the routelet's transport module accordingly. It arrives at the routelet's virtual input port, is processed in the forwarding engine and sent out on the virtual output port. However, it is not sent to the next node yet. It first arrives at the virtual input port of its parent routelet. Here the process is repeated and so on, until it arrives at the root routlet's input port. Here it is processed for the last time and finally forwarded to the next node in the network. This is rather inefficient as the data is touched, not just once, but potentially many times. This is an example of multiplexing at many layers. Contrast this with the "flat" model of netlets. Here netlets form simple partitions of lower-level netlets that can be controlled in arbitrary ways, e.g. by setting up connections from one endpoint to another. The data simply flows through these connections and is not touched at all. Multiplexing happens once and at the lowest possible level.

Intelligent Networks (IN) [25] allow the introduction of new services by associating them with signalling endpoints. Basic calls are separated from IN-based calls. For example, dialling an 0-800 number will trigger a temporary suspension of call-processing and initiate a series of transactions between the Service Switching Point (SSP) and the Service Control Point (SCP), which is essentially a real-time database. A lookup in this database (e.g. for a 0-800 number) tries to find the corresponding *service logic*, i.e. the code which is then executed. The code sends back instructions to the SSP on how to process the call. The bulk of current IN transactions consist of translating the number dialled by the caller into another number.

11 Conclusions

We have discussed the *Haboob*, which represents a direction in network control, called *elastic control*, that combines elements from active networks and open control. It is like open control (and intelligent networks) in that it maintains a separation between control and data and attempts to open up the traditionally closed communication systems, and like active networks in the ease with which new code can be added to existing network control by loading it up in an execution environment known as the Sandbox. It also tries to standardise a computational model and defines the Sandbox SUFI to provide DLAs with a uniform interface for communication, code loading, security and other aspects. Above all, the *Haboob* considers the various aspects of network control independently and enables extensibility and customisability for all of them.

12 Acknowledgements

Thanks to Ian Leslie, Rebecca Isaacs and Richard Mortier of the Cambridge Computer Laboratory for their useful comments on earlier versions of this paper.

References

[1] K. van der Merwe, *Open Service Support for ATM*. PhD thesis, University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., Feb. 1998. Also available as technical report 450.

[2] A. Lazar, K. Lim, and F. Marconcini, "Realizing a foundation for programmability of ATM networks with the binding architecture," *IEEE Journal on Selected Areas In Communications*, vol. 14, pp. 1214–1227, Sept. 1996.

[3] D. Tennenhouse and D. Wetherall, "Towards an active network architecture," *ACM Computer Communication Review*, Apr. 1996.

[4] D. S. Alexander, , M. Hicks, P. Kakkar, A. Keromytis, M. Shaw, J. Moore, C. Gunter, T. Jim, S. M. Nettles, and J. Smith, "The SwitchWare active network implementation," in *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

[5] G. Goldszmidt and Y. Yemini, "Delegated Agents for Network Management," *IEEE Communications Magazine*, vol. 36, pp. 66–70, Mar. 1998.

[6] D. S. Alexander, M. Shaw, S. M. Nettles, and J. Smith, "Active bridging," in *Proceedings of ACM SIGCOMM'97*, (Cannes, France), Sept. 1997.

[7] B. Krupczak, K. Calvert, and M. Ammar, "Protocol boosters: Applying programmability to network infrastructures," *IEEE Communications Magazine*, vol. 36, pp. 93–99, Oct. 1998.

[8] J. Biswas, A. Lazar, J.-F. Huard, K. Lim, S. Mahjoub, L.-F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, "The IEEE 1520 Standards Initiative for Programmable Network Interfaces," *IEEE Communications Magazine*, vol. 36, pp. 64–70, Oct. 1998.

[9] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Proceedings of OPENARCH'98*, (San Francisco, CA), Apr. 1998.

[10] A. Lazar, "Programming telecommunication networks," *IEEE Network*, vol. 11, pp. 8–18, Oct. 1997.

[11] S. Shenker, C. Partridge, and R. Guerin, "Specification of Guaranteed Quality of Service (RFC 2212)," tech. rep., IETF Network Working Group, Sept. 1997.

[12] J. Garrahan, P. Russo, K. Kitami, and R. Kung, "Intelligent Network Overview," *IEEE Communications Magazine*, vol. 31, pp. 30–36, Mar. 1993.

[13] J. W. Stamos and D. K. Gifford, "Implementing remote evaluation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 710–722, July 1990.

[14] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas In Communications*, vol. 14, Sept. 1996.

[15] H. Bos, "ATM Admission Control based on Measurements and Reservations," in *Proceedings of the IEEE International Performance, Computing and Communications Conference*, (Phoenix, Arizona), pp. 298–304, Feb. 1998.

[16] R. Gibbens and F. Kelly, "Measurement-based connection admission control," in *15th International Teletraffic Congress Proceedings*, June 1997.

[17] A. Campbell, H. D. Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela, "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures," in *Proceedings of OPENARCH'99*, (New York, USA), Mar. 1999.

[18] H. Bos, "Application-specific policies: Beyond the domain boundaries," in *Integrated Network Management VI (IM'99)*, (Boston, USA), pp. 827–840, Chapman & Hall, May 1999.

[19] UNI4.0:, "ATM User-Network Interface Specification - Version 4.0," *ATM Forum Document: af-sig-0061.000*, 1994.

[20] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. C. Liaw, T. Lyon, and G. Minshall, "General switch management protocol specification—version 1.1." RFC.

[21] FORE, "SPANS UNI: Simple protocol for ATM signalling, release 3.0," *Fore Systems Inc., 1000 Fore Drive, Warrendale, PA 15086-7502, USA*, 1995.

[22] H. Bos, "Building a distributed video server using advanced ATM network support," in *Proceedings of the second IFIP/IEEE International Conference on Management of Multimedia Networks and Services '98*, (Versailles, France), Nov. 1998.

[23] D. Niehaus, A. Battou, A. McFarland, B. Decina, H. Dardy, V. Siraky, and B. Edwards, "Performance benchmarking of ATM networks," *IEEE Communications*, vol. 35, pp. 134–143, Aug. 1997.

[24] G. Knight and R. Hazemi, "Mobile Agent based management in the INSERT project," *Journal of Network and Systems Management*, vol. 7, Sept. 1999.

[25] ITU-T, "Recommendation M.3010. Principals for a Telecommunications Management Network," *ITU publication*, 1992.

Biography

Herbert Bos holds an MS degree from the University of Twente in the Netherlands and is currently completing his Ph.D. at the Cambridge University Computer Laboratory. His research interests include active networks, agent technology, resource management based on a speculative free-market model, and devolved control of multi-service networks.