Elastic Network Control: An Alternative to Active Networks

Herbert Bos, Rebecca Isaacs, Richard Mortier and Ian Leslie

Abstract: **Much of the recent work in network control has concentrated on opening up the network, moving away from closed, monolithic control systems. Such work can be placed into two categories: use of generic APIs, giving rise to *open control* of networks; and allowing packets on the datapath to program the network nodes, leading to the concept of the *active network*. Drawing on experience in both of these areas, this paper presents the design and implementation of a framework that enables clients to program all aspects of the network in a manageable, controllable manner.**

By using resource partitioning, clients are allowed to inject code into the various entities of the network's control system, subject to access control restraints. This form of programmable network is termed an *elastic network* and it is proposed as an alternative to current active networking approaches. The elastic network has two key advantages over open control and active networks. Firstly, the elastic network allows dynamic customisation of all aspects of the network control, management and data processing components. Secondly, all customisation takes place in a safe, resource-controlled way.

This makes available the flexibility of active networks, and the resource controllability of open control networks, without imposing the attendant costs on all users of the network. Costs still exist, but their imposition is restricted to those places where the available functionality is actually used.

Index Terms: **Programmable networks, active networks, open control, open signalling, network architectures.**

I. Introduction

RESEARCH aimed at making networks programmable has taken two main directions: *active networks* and *open control*. Both of these approaches suffer from limitations. ANs (Active Networks), whilst highly flexible, generally provide poor support for resource control and management, and impose overhead everywhere on the data path.

H. Bos is with the Leiden Institute of Advanced Computer Science, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands. E-mail: herbertb@liacs.nl. R. Isaacs is with Microsoft Research, 1 Guildhall Street, Cambridge, UK, CB2 3NH. E-mail: risaacs@microsoft.com. R. Mortier and I. Leslie are with the Systems Research Group, Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge, UK, CB2 3QG. E-mail: Richard.Mortier,Ian.Leslie@cl.cam.ac.uk.

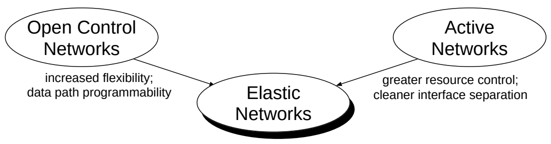
Open Control Networks
increased flexibility; data path programmability
Elastic Networks
Active Networks
greater resource control; cleaner interface separation

Fig. 1. How elastic networks derive from active and open control networks.

Conversely, although OCNs (Open Control Networks) have good support for resource control and clear separation between the data and control paths in the network, they are often very inflexible, being difficult to extend and customise. As depicted in Figure 1, our approach aims to provide the flexibility of ANs, whilst retaining the resource assurances and controllability of OCNs.

This section introduces current approaches to network programmability and describes how the work presented in this paper is related. Section II presents a brief introduction to network programmability and discusses related work. Section III reviews the context of this work and motivates the elastic network as a solution to the problems encountered in our earlier research on open signalling and control. The main contribution of the paper is presented in Section IV, where a general framework for providing extensibility is described and explored. Examples of use are given in Section V and conclusions drawn in Section VI.

A. Active networks and open control

An AN allows programmability of internal network nodes. An AN node is a network node that runs a NodeOS (Node Operating System), responsible for the partitioning of resource and other shared functionality. Such a node provides one or more EEs (Execution Environments) within which programs are executed, each EE being allocated a resource partition on the node. Within an EE, code may be introduced as a program encapsulated in the received data packets (so-called 'capsule based' or 'pure' ANs), or it may be downloaded onto the node prior to the arrival of the data on which it will operate. An active network is *smart*, in that its deployment places sophisticated computational capabilities in the core of the network. Consequently, care must be taken to ensure the security and integrity of code introduced into the network, and use of resources must be strictly controlled to guarantee fairness.

In contrast, an OCN moves intelligence out of core network elements, and provides a resource abstraction together with an open programmable interface through which network resources may be manipulated. Unlike active networks, programmability is typically available only on the control path, although some OCNs do allow computation on the data path. Programmability is provided via the introduction of new services and control systems. The major difficulty with this approach is that of defining a resource abstraction that is sufficiently flexible, expressive, and general to allow new and innovative network services and network technologies, and yet practical to implement.

B. Elastic networks

We term the approach of this work to the provision of network programmability *elastic networking*; it incorporates aspects of both active and programmable networks. Starting with a powerful OCN, discussed briefly in Section III-A and a simple mechanism for dynamically loading code, we considered how to merge these two approaches. The result is a highly flexible implementation of an elastic network. This makes two key contributions: firstly, it allows dynamic customisation of the network control, management and data processing components at all levels of granularity and timescale; and secondly, all customisation takes place in a safe, resource controlled manner. The second point is crucial and is achieved by hard partitioning of the network resources. This enables clients to be given control over 'raw' resources if they desire, while still guaranteeing that their actions cannot interfere with the resource guarantees of others.

To enable partitioning, we define a resource abstraction through which control operations can be performed on network nodes, and provide a framework which permits the dynamic creation, modification, and deletion of VPNs (Virtual Private Networks). Each VPN can manipulate its allocated resource partition as desired through an open control interface, and the owner may thus exert complete autonomous control over the VPN. Programmability in this manner allows the VPN owner to run the network control and management system of their choice, examples including IP, MPLS, PNNI and control systems optimised for a particular network service such as video-conferencing. Partitioning allows many such VPNs to co-exist on the same physical network, each running an independent control system using a dedicated subset of the network resources.

At the same time, we also allow for functionality deriving from the AN approach. Rather than limiting programmability to the network resources themselves as in open control systems, or to packets on the data path as in most active networks, the elastic network strives to allow clients to extend and override network functionality at many points of both control and data planes, and at many different timescales. This allows network clients freedom to choose their own trade-offs, in a safe and controllable manner. A key property of this approach is that the timescale of influence is not restricted to either packet or flow creation timescales as indicated by Figure 2.

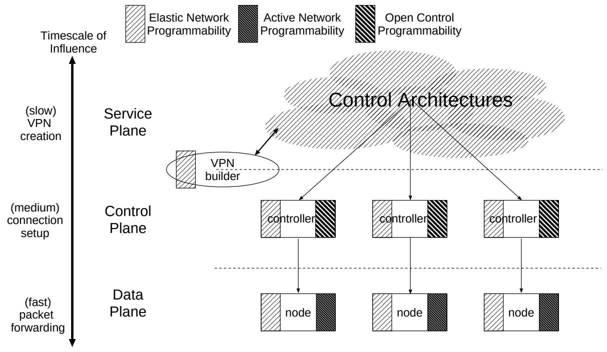

Fig. 2. Timescales of influence.

II. Network programmability

This section first presents various properties which may be used to characterise network programmability, and then describes related work.

A. Characteristics of network programmability

The facility for programming the network can exist throughout the network's data, control, and management planes. It can operate at varying granularities within those planes and may be categorised according to whether:

1. *Code is static or dynamic*: Static code generally comprises processes initiated manually on certain nodes. These processes often consist of native code, compiled for a particular target platform, such as signalling software installed on a traditional switch by a network administrator. Conversely, dynamic code, termed DLAs (Dynamically Loadable Agents), can be added to and removed from the network on the fly, using suitable code loading mechanisms. Examples of such dynamic code techniques are found in agent technology and active networks.

2. *Functionality is fixed or extensible*: The distinction between static and dynamic programmability is independent of its fixed or extensible nature. For example, a statically programmed network (e.g. an OCN) is still extensible through the loading of DLAs into the nodes. In other words, it is possible to extend static functionality with dynamic code, with DLAs able to extend or even override the functionality of the fixed code. Alternatively, static functionality may be *fixed*, with the complete functionality of a component determined at compile time. In the former case, safety and access control are important issues, as one does not want to give access to powerful and potentially disruptive operations without suitable precautions. However, it may well be a requirement that at least the *operators* be able to control the network as they see fit.

3. *Code is installed through the data path or through some other means*: Code to extend the network programmability model may be installed over the channel that the data travels, or over a separate channel. The former approach is advocated by the (capsule-based) active networks community: an *active packet* or *capsule* is sent to a network node *in-band*, where it is demultiplexed from the data by some means and then executed in the appropriate EE. Such code is almost always dynamic. The latter approach keeps the data and programming channels completely separate, either physically or logically. For example DLAs may be transported over a completely separate network, or simply a separate connection with a specific resource guarantee. Making such a distinction between code and data has many desirable properties. For instance, even in times of congestion it remains possible to send programs to deal with the congestion into the congested area of the network. It also allows better control of the amount of bandwidth and other resources consumed by the code.

4. *Computation is performed in the control and management planes, or in the data plane*: Related to the manner in which the code is downloaded is the question of what the code may act upon. In the extreme case computation may be performed upon individual data packets, as advocated by the AN community. This has the disadvantage of introducing computation on the data path which tends to slow data transmission, but does allow features such as filtering and transcoding to be installed in a very straightforward manner. Alternatively, computation may pertain only to flows or connections. For example, code may create or delete connections across the network without ever touching the data itself. This approach falls entirely within the control and management planes, and has the advantage that there is no interference with data transmission: packets are switched at full speed and entities on the data path need not be aware of new or modified functionality. A similar but more extreme example is to allow code to act on 'large' constructs, such as VPNs. Such code could customise the way in which VPNs are constructed and evolve.

5. *Code runs inside or outside the core of the network*: This final categorisation concerns the location of the computation. This can either be in the core of the network, with clients running their own code on the network operator's nodes, or it can be on clients' nodes themselves. In the former case safety and security must be ensured, to prevent one client maliciously interfering with another. In the

latter, code may be run on clients' nodes, and only interact with the network through a well-defined API, leading to fewer security risks. ANs are an example of the former, while a control process running on a client's machine, invoking methods on an open network API is an example of the latter.

B. Current approaches to network programmability

We argue, as do the advocates of OCNs, that computation on the data path should be minimised, and should certainly not be the default method of extending network operation. Most packets should simply be switched toward their destination. This suggests that computation should typically pertain only to control and management tasks. However, we do recognise, as do the proponents of ANs, that computation on the data path can be very useful in certain areas such as transcoding and filtering, as already mentioned. Therefore such functionality *should* be available in any network programmability model.

Applying the model from Section II-A, we see that what are commonly known as ANs are usually characterised by dynamic code loaded on a static EE, extensible functionality, code installation through the data path, and computation in the data plane and on the network nodes themselves. On the other hand, OCNs are generally characterised by static code with largely fixed functionality, programmability installation through explicit signalling, or even operator intervention, computation pertaining principally to the control and management planes, and by running outside of the core of the network.

Consequently, our goal is to enable programmability anywhere in the spectrum between these two approaches, wherever it is considered useful, but without imposing penalties (e.g. performance degradation) on applications or networks that do not need the functionality. Although our focus is on control and management, this paper discusses a framework which enables all the different types of network computation mentioned. Elastic networks aim to provide flexibility in all of the areas mentioned, permitting any feasible combination of properties. In order to do so, they also require the proper enforcement of resource control throughout the network.

C. Related work

The SwitchWare [1] project covers a number of aspects of an AN architecture, addressing security and protection concerns in particular. The architecture comprises three layers: at the lowest layer is a secure active router which guarantees the integrity of the underlying system to the upper layers. The middle layer consists of potentially dynamically loadable *active extensions*, written in CAML, providing network services to the *active packets* of the highest layer. Active packets contain both code and data, and are written in a programming language, PLAN, that has the necessary type safety and resource limiting mechanisms. Although general-purpose AN architectures such as SwitchWare provide some degree of network programmability, their application is essentially limited to operations on the data path. Other aspects of network service provision such as resource partitioning and virtual network provisioning are not addressed by such approaches. In contrast, programmability is available throughout our model, with flexibility in every aspect of the Tempest infrastructure.

Application of AN mechanisms in order to introduce or augment network protocols is the concern of both ANTS [2] and Protocol Boosters [3]. ANTS is an AN toolkit for building and deploying new network protocols. Protocol code is distributed inside capsules to active nodes where it is executed. Execution takes place within an execution environment that restricts the primitive operations available and the amount of resource that can be used during processing. The degree of programmability is necessarily limited by the primitives that are available within the EE, dictated by the particular requirements of the forwarding routines. Protocol Boosters allow the functionality of a protocol to be extended on the fly. An example is the addition of FEC (Forward Error Correction) to the data path when it is discovered that one link is very 'lossy.' The effect of the booster is restricted to that particular link, so computation is added only where required. Section V-E shows how such enhancements were implemented in an elastic network.

Network management and monitoring is another application of network programmability under active investigation. NetScript [4] defines a language for programming network node functions such as routing, packet classification and signalling. Mobile agents are dispatched throughout the network to execute NetScript programs on streams of packets. Smart Packets [5] exploit the functionality of an AN to move network management and monitoring tasks closer to the network nodes, to customise the information returned, and to automate problem resolution where possible. The benefits obtained by such specialised uses of ANs also apply when similar functionality is implemented in the Tempest, as described in Section V-B.

The IN (Intelligent Network) architecture incorporates a limited form of programmability into the traditional telephony network. Call state models are maintained at the telephone switches, known as SSPs (Service Switching Points), with trigger detection points that are activated by certain events such as the customer dialling a particular 4-digit number. Control messages are then transmitted to SCPs (Service Control Points) which contain the logic and databases for available intelligent services. More sophisticated functionality, such as voice-activated processing, can be incorporated in an intelligent peripheral attached to an SSP or SCP. IN services are somewhat cumbersome and complex to deploy, and are limited by the expressiveness of SS7's control messaging protocol. As an example, an implementation of an IN, known as INCA [6], was built on an elastic network. INCA is described in Section V-C.

The Multiservice Switching Forum [7] is a group of telecommunications companies, including equipment manufacturers and network service providers. The stated goal of the group is to define an architecture separating control and data planes that facilitates the introduction of new network services over ATM-capable networks. To that end an open switch control interface, together with implementation agreements to ensure interoperability between components from different vendors, is being defined. The SoftSwitch Consortium [8] has similar goals in the IP arena. Other bodies attempting to define network programmability APIs are Parlay [9] and the IEEE P-1520 standards project [10].

Netlets [11], [12], are a way to recursively repartition resources in a network. Their nature and the manner in which they are related to elastic networks will be discussed in Section V-B. Other work using netlets includes Virtuosity [13], which is a virtual network resource management system that *spawns* child virtual networks. Spawning refers to the automated creation, deployment and management of virtual networks. Resource allocations to spawned virtual networks are enforced with measurement-based admission control, with a limited set of traffic classes supported to ensure virtual network independence. This contrasts with the control plane policing employed in the Tempest, which does not constrain the operations of a control system beyond the limitations already imposed by the switch control interface. Furthermore, using the elastic control methods described in this paper even these limitations to a large extent can be alleviated. The Tempest relies on the network hardware to carry out in-band policing and shaping to ensure the proper behaviour of traffic belonging to a VPN. A control system providing netlets in an elastic network is described in Section V-B.

III. Context

The work presented in this paper builds on the *Tempest* [14], [15], a network control framework based on the *switchlets* concept. This section briefly reviews this framework, and identifies those limitations that motivate the work on elastic control. A mature implementation of the Tempest currently exists, and is in use at Cambridge University Computer Laboratory, and at BT Adastral Park. At Cambridge University Computer Laboratory it runs on an ATM network comprising 5 switches, several hosts and 4 video codecs. In the past, the focus of work on the Tempest has been the particular implementation for ATM networks. However, it should be noted that the approach of the Tempest and many of the lessons learned from its implementation are not restricted to ATM networks. To support this claim, an implementation over MPLS has been performed and is reported elsewhere [16].

A. The components of the Tempest

The Tempest is an implementation of a framework for network control based on the idea of partitioning network resources at the lowest possible level. As described in Section I-A, telecommunication networks' control systems have traditionally been monolithic and closed, leading to numerous problems with their control and management. The Tempest addresses these problems by allowing multiple independent service providers to be present in the same physical network and by providing a reduction in the time taken to develop and deploy new services in the network [14].

We identify six principle components forming the Tempest environment. At the lowest level of the architecture is the *switch control interface*, operating between the control software and the physical switch. In traditional telecommunications systems this interface is closed and its functionality is tightly coupled to the network hardware. On the other hand, our implementation of the Tempest uses an open, generic switch control interface called *Ariel*, allowing for non-proprietary, non-standardised network control systems. Ariel is sufficiently simple and portable to be implementable for even the most basic switches. Other such interfaces exist, such as Cisco's VSI (Virtual Switch Interface), and the IETF's GSMP (Generic Switch Management Protocol), likely to be standardised by the Multiservice Switching Forum. It should be noted that it is perfectly possible to have a Tempest implementation which allows many types of switch control interface to be exported, provided that all can be mapped onto the underlying interface that interacts with the physical switch.

The second component is the *divider* process, enabling the partitioning of network resources. One such process exists for each switch in the network. It resides on or off the switch – in both cases providing the same functionality, but with differing performance. It sits between the interface to the control system, and the interface to the physical switch, and polices each invocation. It is therefore capable of enforcing guaranteed, 'hard' partitions of the switch resources, termed *switchlets.* The resources each owns are then manipulated through an instance of the Ariel interface given to the control system. Consequently it appears to the software that it is independently controlling its own complete, albeit smaller, switch. This partitioning allows multiple control systems to be concurrently active on a physical network, and guarantees that they cannot interfere with each other's network resources.

The third component is the *netbuilder*, the process responsible for combining switchlets to create VPNs. As they are collections of switchlets, such VPNs are hard partitions of the *network* resources and each appears as a smaller physical network.

The fourth component is the CA (Control Architecture), consisting of the set of protocols, policies and algorithms used to control (a partition of) the network. This is the software that sets up, tears down, and otherwise manages connections, and in short, actually controls the VPN. One of the principle motivations for the original work on the Tempest was the belief that there can be no such thing as a one-size-fits-all control architecture. Many CAs exist today, such as Q.2931, RSVP, PNNI, and IP Switching, all meeting different requirements, but it is unrealistic to expect any of these to evolve into the ultimate control architecture catering for all needs, past, present and future. The Tempest permits a number of different CAs to be simultaneously active on the same physical network. For example, Figure 3 shows three CAs simultaneously active on two switches. One of them controls two switchlets on two switches. The Tempest ensures the following property: whatever happens in these switchlets will never interfere with the other two switchlets, i.e. each CA's operations are limited to its own partition of the resources.

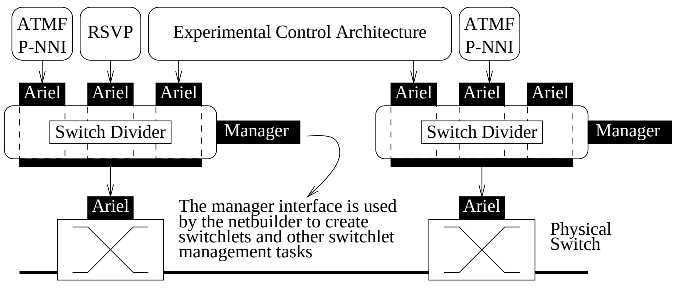
ATMF P-NNI
RSVP
Experimental Control Architecture
ATMF P-NNI
Ariel
Switch Divider
Manager
The manager interface is used by the netbuilder to create switchlets and other switchlet management tasks
Physical Switch

Fig. 3. Multiple Control Architectures running independently on a single physical network, partitioned by dividers.

The usual sequence of events in the current implementation of the Tempest is as follows. At start of day the CA requests a VPN from the netbuilder. The netbuilder in turn requests the corresponding dividers to create the appropriate switchlets. The dividers do so, returning a reference to a new Ariel instance for each switchlet created, which is then returned by the netbuilder to the CA. The CA subsequently controls its switchlets directly by invoking Ariel methods to control creation, resource allocation and deletion of connections. Finally, the CA makes a call to the netbuilder to destroy the VPN causing the dividers to destroy the switchlets that were owned by the CA.

The fifth and sixth components are the *generic service* and *data path* components. An example of a generic service is a federated trading mechanism used to locate services such as the netbuilder. Data path components are elements that reside on the data path and participate in network control. Examples include filters, transcoders, and other data processing engines.

IV. Programmability in the Tempest

The basic Tempest model of low-level resource partitioning coupled with open devolved control has a number of advantages over traditional network control, but suffers from limitations. Foremost amongst these is that it is not truly open: components are themselves closed and provide rigidly defined functionality. This limits the manner in which resources may be partitioned and VPNs built. In addition, there will be vendors who wish to differentiate themselves based on the control functionality they can provide to clients, whilst not wishing to build custom CAs from scratch. Also, there are situations where functionality is most appropriately provided dynamically, and where performance may be improved by adding extensibility. This section describes how programmability was added to the Tempest to enable this.

A. Adding elasticity

Elastic control allows control and management functionality to be customised on a per-network, per-virtual network, per-switchlet, per-connection, or even per-packet basis. Fundamentally, the environment in which control and management functionality is implemented is standardized, rather than the control and management functionality itself. This concept is also important in active networks, where the computational model, comprising the instruction set and resources available for computation on or by packets as they pass through active network nodes, is standardised rather than the set of operations that may be performed on or by packets passing through the node [17]. In active network terms, our model extends the scope of the execution environment of an active network node from the data plane to the control and management planes.

Our solution allows programmability at both the network edge and core, and gives maximum flexibility to those implementing and deploying CAs, *whilst retaining the advantages of control and protection of resources that the Tempest offers.* Secure, authenticated access to the programmability interface is clearly necessary, and will be considered following a more detailed discussion of our model. More detailed evaluation is available elsewhere of the Tempest [14], [15], of Noman and other sandboxes [18], and of Pyman and INCA [6]. In general, control operation timings are on the order of standard signalling protocols.

B. The sandbox

The basic building block for elastic control is a safe execution environment – the *sandbox.* Pieces of mobile code, known as DLAs, may execute inside sandboxes situated throughout the network control framework. All aspects of the Tempest have been extended with sandboxes and can therefore be customised. DLAs may be *active* or *passive.* Active DLAs consist of code that begins execution immediately (for example, an iteration over an information base), whereas passive DLAs install themselves in the sandbox but are not activated until explicitly called (for example, as a 'function' that will iterate over an information base when called). The sandbox is language and implementation independent as only its interfaces are specified.

Every sandbox exports the SUFI (Simple Uniform Framework for Interaction), which provides operations that allow DLAs in different Sandboxes to communicate, whilst preserving location transparency, either via remote evaluation [19], or via more traditional RPC. These SUFI operations are extremely simple. For example, parameters in remote evaluations are treated as byte sequences and SUFI relies on *adaptors* to do the marshalling, unmarshalling and type checking of the parameters at runtime. Adaptors are small pieces of dynamically loadable code able to parse a parameter string at a remote site, extract and structure these parameters in the appropriate way, and return the structured data. Remote evaluation of code (and state) combined with the ability to remove such code from a node provides a way to implement replicating and migrating DLAs. The SUFI also provides for: export of the interfaces of a DLA, access control to a DLA's operations, authentication and trust establishment, reflection, and mechanisms (threading and semaphores) for interaction between DLAs.

Once loaded, DLAs can export their interfaces to other DLAs and thence to the outside world using the SUFI, allowing clients to extend the functionality of the sandbox, the component containing the sandbox, or both. This allows each of the Tempest components to be customised. Figure 4 illustrates how modules can be added in a layered fashion to a sandbox by DLAs, with modules at a higher layer dependent only on those at lower layers. The sandbox itself might sit on a network node or host computer. In the situation on the left a DLA implementing a transcoder module is loaded onto a node that already contains a number of other modules. In this example, the transcoder only uses functionality provided by the module called 'base layer,' so it is placed directly above the base layer on the node, next to the SUFI module. Next, a new module is loaded, containing a quality adaptor. In the example, this module depends on the functionality of both the transcoder and the application-specific (AppSpec) modules. The resulting layering of modules is shown in the rightmost diagram of Figure 4.

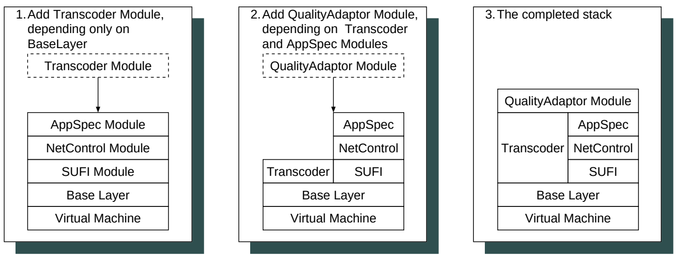

Fig. 4. Extending the functionality of a sandbox using layered modules.

All components of the Tempest have been provided with an operation that allows clients with the appropriate access rights to request the instantiation of a sandbox on control architectures, netbuilders, dividers and implementations of the Ariel interfaces (and even on end-user applications, if needed). Concretely, realising a sandbox simply requires a single C++ class to be instantiated. The SUFI module is automatically installed in each of the sandboxes, and then specialised modules are loaded for each of the components, allowing DLAs to customise that particular component.

C. Safety and resource control

Considerable thought has been given to safety and security. Careful shielding, strict access control, trust establishment, and controlled access to resources are all part of the sandbox implementation. In addition to utilising the Tempest framework to control access to network resources, DLAs should also be prevented from using too much CPU time, memory, and other contended resources on network nodes. This can be enforced by the use of a resource controlled operating system as the NodeOS, such as Nemesis [20], which allows the partitioning of resources inside the operating system. RCANE [21] has demonstrated that it is possible to support active network environments on top of Nemesis in a safe and resource-guaranteed way.

In contrast with other programmable network proposals, including active networks and IN, our model enables us to dynamically customise all levels of network control, from the switch control interface up to the CA, and even to end-user applications. Programmable control operations can thus be applied to, for example, protocol integration and evolution, service development and deployment, routeing, congestion control, network management and maintenance.

V. Examples of Use

We have described how programmability has been introduced throughout the Tempest framework. This section presents some example applications to illustrate the increased flexibility and utility of our approach. We have implemented all of them.

A. Ariel DLAs

As an example of such a component-specific module, a sandbox for an Ariel implementation is provided by a module named `DLariel` which allows DLAs to call all the standard Ariel operations (e.g. to set up or delete a connection across a switch, or to gather statistics). Furthermore, such DLAs may implement entirely new functions as well. These functions may be built on 'standard' Ariel operations. For example, to set up or delete a large set of connections, a DLA might provide a single function which translates to a large number of Ariel operations. In this case, there will be a performance gain as remote clients need only call a single function over the network, instead of making a call for each connection. This is an example of a passive DLA. An example of an active DLA might be a 'remote monitor' DLA which uses Ariel methods to automatically gather statistics from a switch on behalf of a CA, correlates the data locally, and periodically sends information (e.g. aggregate traffic measurements or alarms) to a remote management site.

Alternatively, DLAs may implement functions that are built on what are known as *native* methods, which are not part of the standard Ariel repertoire, but are supported by the switch. For example, if the Ariel interface is implemented over GSMP, the switch will support a `MoveBranch` operation, which is not part of Ariel. These operations may be exposed to DLAs as native methods. A DLA may then export this `MoveBranch` operation to other clients. Note that to be portable the DLA may first check to see whether such a native method is available; if not, the DLA then simply falls back on the standard Ariel operations.

`DLariel` even provides users with the appropriate access rights the ability to *override* some of the standard Ariel operations. A trivial example of this would be to override the operations to create or delete connections with the empty operation. Installing this for a switchlet (e.g by the system administrator) allows the controlling CA subsequently to execute any Ariel operation it wishes (e.g. to gather statistics), except those that create or delete connections.

It would be difficult to achieve such functionality in an AN, due to their lack of distinction between control and data planes. Whilst it is easy in an AN to specify the packets to which computation will apply, there is little notion of explicit control operations. On the other hand, OCNs, whilst providing a clear data/control separation, do not allow general customization of existing components, such as the switch interface. Although such a facility could be retro-fitted, it would be difficult to do in a suitably efficient and flexible manner. By building it in from the start, as with elastic networks, such problems are avoided.

B. Elastic CAs

`DLariel` allows one to customise access to a switch but the CA itself can also benefit from programmability.

Highly advanced functionality can be implemented using `DLariel`, to the point where the CA itself is just a DLA running in the divider. However, this will normally not be the case for a variety of reasons: not only is it relatively inefficient to run an entire CA as an agent, particularly if the agent consists of interpreted code, but network providers might also simply not allow sufficient access to run code in their dividers. The expected behaviour is to have a CA (probably consisting of at least some *static* code), communicating with an implementation of the Ariel interface, as shown in Figure 3.

However, extensive programmability can still be provided by making the CA itself programmable. An advantage of doing so is that the CA generally supports richer functionality than the very basic Ariel operations, such as advance reservations, and this can then be made available to DLAs. In other words, the original static functionality of the CA can be viewed as a library for the DLAs running in its sandbox. Another advantage of a programmable CA over a simple `DLariel` interface is that the latter only concerns a single switch, whereas a CA may control any number of switches. Hence, CAs may offer end-to-end functionality across a number of switches, including such features as call-backs to applications, beyond the scope of an Ariel implementation. In the remainder of this section, we will discuss a CA which offers extensive programmability.

We have equipped this experimental CA with functionality that allows clients of the CA to create and control *netlets*. Netlets are sub-partitions of a VPN that can themselves be controlled arbitrarily by clients, under the auspices of the owning CA. Netlets can be dynamically created, modified, deleted and even repartitioned, in the same manner as VPNs. One of the problems netlets address is which control primitives for which types of connection to provide to clients. Whichever set of end-to-end primitives is chosen, it is unlikely to be sufficient for all possible uses, present and future. The solution provided by netlets is to enable clients to control their resources at a much finer level of granularity, without requiring that they build a CA from scratch. For this purpose, low-level primitives are provided to enable clients to control the individual resources of a netlet, e.g. to create arbitrary connections on any single switch in the netlet, or to request statistics for a particular switch port. Clients can then build their own functionality, such as their own connection types, *within* an existing, full-fledged CA.

The problem with allowing remote clients to exercise control at a very fine level of granularity is the communication cost involved. For example, using just the fine-grained control primitives, it can take many calls across the network from a remote client to the CA to set up an end-to-end connection. In the Tempest, this problem has been solved with DLAs. In particular, the low-level netlet primitives are accessible to DLAs running in a CA sandbox. Rather than making a separate request for each of the individual operations, a client installs a DLA, to make these requests locally. The CA still invokes the primitive Ariel operations on the switches concerned, but the communication between remote clients and CA is reduced to the downloading of a single DLA, rather than numerous Ariel invocations. This allows users to efficiently implement application-specific functionality. When combined with `DLariel` interfaces, extended to reduce the communication overhead between CA and divider, this can dramatically reduce the overheads of setting up application specific connection types. In this way, the combination of netlets and DLAs generalises the concept of ‘connection closures’ [22].

A more ‘VPN-like’ example of the functionality provided by netlets is *differentiation of policing*. In general, it is important that VPNs are policed *in-band*, since misbehaviour in one VPN cannot be allowed to affect the behaviour of others. However, using netlets the strictness of policing can be relaxed if, for example, the applications in the netlet are trusted. Hard partitioning between VPNs ensures that even if connections in a netlet do misbehave, the problems will be confined to the containing VPN and not propagate beyond it. Loose netlet policing may be performed by periodically taking measurements from switches to see if the netlet has exceeded its allocated bandwidth. This *looseness* may vary from netlet to netlet, and some netlets may not be policed at all. We built a programmable traffic server which can be instructed by DLAs running in its sandbox to take measurements from a switch and notify the CA when certain thresholds, such as allocated bandwidth, are exceeded.

Netlets have also been used to partition the resources of different types of connections, such as current and future reservations to prevent all resources being reserved in advance. Finally, Bos [12] shows that netlets and DLAs have been successfully used in novel implementations of interoperability between incompatible domains. The idea is that a user-specified DLA may determine for a particular netlet how the primitives of one particular domain are mapped onto those of a neighbouring (incompatible) domain.

As previously noted, ANs do not separate control and data, so they have no explicit notion of a CA. ANs may however provide customised functionality in the network nodes, e.g. allowing clients to specify their own routing functions. Resource partitioning has not been a particular focus of AN work and such things as netlets do not map naturally onto ANs. OCNs do have a strong notion of the separation of control and data and of resource partitioning. However, CAs in OCNs are static entities with rigid functionality and must be taken down when changes are required.

C. Noman and Pyman

Noman and *Pyman* are templates for CAs that enable clients to upload their desired CA functionality and then modify it on the fly. In other words, they are not CAs in and of themselves, but rather template CAs, allowing one to build, or load, and then modify a CA rapidly and dynamically. They are implemented by extending, respectively a Tcl or Python, sandbox with a network control module. This module provides DLAs with a simple API which allows them to create, modify, control and destroy virtual networks. In the current implementation Noman can also be used 'in' the network, in the same process as the divider, whereas Pyman always exists at the edge of the network. This distinction demonstrates two different locations where network programmability may be provided, giving rise to different performance trade-offs. Noman allows for *dynamically loadable* CAs, where (potentially remote) clients with the appropriate access rights are able to dynamically load their own CA and then to add, delete or modify functionality while the CA is in operation.

Such a CA template concept is attractive for a variety of reasons. Firstly, it provides a rapid prototyping tool for the development of new CAs, offering a programmable interface to the Tempest components. Secondly, it can be used to combine libraries of various standard CA components, e.g. routing, connection setup, connection admission control, to create the desired CA. This is related to the way that components are bound together in *xbind* [23], but is more general as components can be customised and replaced on the fly. Thirdly, it allows applications to customise their CA.

An example of a system that makes use of Pyman at the edge of the network is INCA [6], a CA that mimics a traditional IN architecture, while taking advantage of the flexibility of the Tempest to offer value-added IN services. It makes use of the programmable environment to allow new IN services to be introduced and existing ones to be upgraded whilst active, without disruption to existing calls in the network. Traditional IN services such as 800 number translation are easily incorporated, and it is similarly straightforward to introduce novel services such as advance call reservation.

Noman has been employed for a simple experiment in bandwidth management based on a speculative free-market model for optimal resource allocation. Clients in this system are charged for used bandwidth per time-unit at the current market rate. The CA, Noman-Auction, allows for two types of reservation: *fixed bandwidth*, which guarantees the bandwidth that an application receives regardless of cost; and *fixed price*, where the client always pays the same price, but the bandwidth they receive fluctuates. However, it is also possible to 'speculate' on the bandwidth market. A client might decide to reserve a large amount of bandwidth when the price of bandwidth is low. It will have to pay the market rate per time unit for this speculation, but this may be a good investment, as the client is allowed to 'sell' the bandwidth when it becomes scarce and hence expensive.

Noman-Auction allows client A to sell bandwidth to client B through a one-time transfer of funds (or credits) from B to A, and the transfer of the bandwidth allocation. After the sale, B will be charged the market rate per time unit. Special operations allow clients to offer bandwidth for sale, and to buy bandwidth for the advertised price. Elastic behaviour is introduced by allowing both buyers and sellers to load DLAs to take care of their bandwidth management policies, e.g. to buy, sell, or reserve bandwidth, depending on factors such as the market rate, 'special offers,' or time of day.

D. Resource revocation

Within the Tempest framework there are two distinct granularities of resource management. Firstly, a CA performs management of the resources *within* its VPN. At the lowest level this is realised by associating some quality of service with the VPN's connections. Secondly, the netbuilder is responsible for allocating resources *between* VPNs. The total resource comprising a VPN is defined by its set of switchlets, so resources can be reallocated simply by modifying the relevant switchlet specifications.

A particularly useful type of dynamic resource reallocation is revocation of resources from a VPN. An example of the flexibility afforded by this feature is the ability to differentiate between 'guaranteed' and 'best effort' VPN resources, with the best effort resources liable to be revoked at any time. Using this knowledge, CAs can offer differentiated service to their applications, while being charged appropriately for the non-guaranteed resources. Another potential use includes maximisation of global resource usage by shifting resources from one VPN to another to cope with short term surges in demand. Resource revocation may also be used to deal with SLA (Service Level Agreement) violations in cases when in-band traffic policing is not sufficient, and to support the withdrawal of resources required for periodic SLAs.

The mechanism by which a resource reallocation takes place is essentially the same as an initial resource allocation to a VPN. This allows resources to be shifted between VPNs with the same degree of granularity, and makes it possible to identify precisely which resources are being shifted. Examples include a specific label range, or the bandwidth currently in use by a particular connection. This information can be exploited by a control architecture to its advantage, as the CA can respond to advance notification of impending resource revocation, and can also suggest a specific set of resources to be revoked.

We have implemented a revocation protocol by which CAs can customise revocation behaviour. As part of the protocol, the netbuilder maintains a list of resources in a *revocation vector* for each CA, representing that CA's preferred order of revocation. The CA, or an agent operating on its behalf, can update the vector at any time. This allows the CA fine-grained control over the resources revoked and enables it to respond appropriately to external events, while still allowing the network operator to revoke resources when required. Victim CAs are notified of the impending revocation in advance of the event, and following the revocation event, the netbuilder will inform affected CAs, detailing which resources have been revoked.

The basic revocation protocol allows a limited form of programmability to be applied to resource revocation. CAs maintain their vectors according to local factors such as application usage patterns or time of day policies. This functionality can be further enhanced by the incorporation of a sandbox in the netbuilder, allowing the CA to express revocation policies that rely on external information it has no direct means of determining. For example, withdrawal of a service offered by a second CA over a VPN on the same physical network might result in the first CA no longer needing some portion of its own resource partition.

The elastic revocation protocol enables a CA to influence the local effects of a global resource reallocation event. Few other existing AN and OCN systems support dynamic resource reallocation between VPNs, and none currently offer customisation of resource revocation. This mechanism supports the flexibility of the system as a whole, whilst enabling the CA to retain its fine-grained VPN resource management capabilities.

The problem of resource revocation in ANs has not been addressed. The notion of a VPN in an AN derives from the IP notion of a VPN and hence usually describes connectivity between end-points, without consideration of resource guarantees [4]. Were ANs offering resource guarantees to be built, consideration would have to be given to resource revocation. OCNs will implement resource revocation much as described above, but CA specific policies would have to be implemented as static components, and communication between these (distributed) components would introduce extra latency and traffic.

E. *Active Networks in the Tempest*

As previously stated, in-band computation does have value in some cases. Using programmability in the Tempest, it is possible to build nodes providing such a facility as and when necessary. An example is the case of the protocol booster described in Section II-C. When a link is unreliable and needs FEC, we dispatch a DLA to reroute the existing connection to a control entity near the link. The control entity adds the FEC to the data by means of a DLA running in a sandbox, and returns it to the data path. On the other side of the link a DLA removes the FEC again. To the endpoints such enhancements of the protocol are completely transparent. The protocol booster mechanism is illustrated in Figure 5. Note that this does not differ in any way from the protocol booster described; we have, in effect, built a very simple (off-switch) active node.

Protocol boosting is a natural example of a use for ANs. However, without mechanism to facilitate dynamic code loading there is no way to support similar functionality in OCNs. Elastic networks show that careful combination of both approaches enables us to have the advantages of ANs (in this case dynamic computation on the datapath), while retaining the resource control and separation of control and management as provided by OCNs.

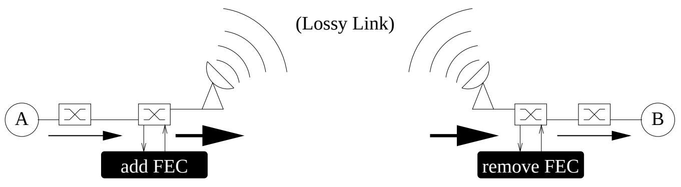

Fig. 5. Active protocol booster in the Tempest.

VI. Summary

This paper presented an overview of our work in the field of network programmability. We gave a model for characterising network programmability, and applied it to the related work. The context in which the work on elastic networks was carried out was summarised, and the shortcomings of the Tempest, our existing resource controlled network control framework were described. It was then explained how programmability was introduced into the Tempest, giving an implementation of elastic network control. Dynamically loadable code executes in a safe enviroment and is provided with modules to allow for interaction and customisation. We concluded by giving a variety of examples of use: `DLariel`, netlets, template CAs, resource revocation, and implementation of active networks within the Tempest.

The work reported in this paper has shown that elastic network control successfully merges the approaches advocated by both the active networks and open control communities. We believe that the resulting elastic network control paradigm provides an efficient, highly flexible system for controlling and managing current and future networks.

Acknowledgements

We gratefully acknowledge the support of BT, Cplane Inc., Marconi, Nortel, Sprint, and Sun, in carrying out this work. We would also like to acknowledge the contributions of Sean Rooney and Jacobus van der Merwe, for the original Tempest concept and implementation work.

References

[1] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network Magazine*, 12(3):29–36, May 1998.

[2] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the First Conference on Open Architectures and Network Programming (OPENARCH'98)*, San Francisco, USA, April 1998. IEEE.

[3] William Marcus, Ilija Hadzic, Anthony McAuley, and Jonathan Smith. Protocol boosters: Applying programmability to network infrastructures. *IEEE Communications Magazine*, 36(10):79–83, October 1998.

[4] Yechiam Yemini and Sushil da Silva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'96)*, L'Aquila, Italy, October 1996.

[5] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart packets for active networks. In *Proceedings of the Second Conference on Open Architectures and Network Programming (OPENARCH'99)*, New York, USA, March 1999. IEEE.

[6] Rebecca Isaacs and Richard Mortier. INCA: Support for IN using the Tempest. In *IEEE Global Telecommunications Conference (GLOBECOM'99)*, pages 812–816, Rio de Janeiro, Brazil, December 1999.

[7] MSForum. Multi-service switching forum. `http://www.msforum.org/`, 2000.

[8] The SoftSwitch consortium. `http://www.softswitch.org/`.

[9] The PARLAY group. `http://parlay.msftlabs.com/`.

[10] IEEE. P-1520: Application programming interfaces for networks. `http://www.ieee-pin.org/`, 1998.

[11] Herbert Bos. ATM admission control based on measurements and reservations. In *Proceedings of the IEEE International Performance, Computing and Communications Conference (IPCCC)*, pages 298–304, Phoenix, Arizona, USA, February 1998.

[12] Herbert Bos. Application-specific policies: Beyond the domain boundaries. In Morris Sloman, Subrata Mazumdar, and Emil Lupu, editors, *Integrated Network Management VI*, pages 827–840, Boston, USA, May 1999. IFIP & IEEE, Chapman & Hall.

[13] Andrew T. Campbell, Herman G. DeMeer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, April 1999.

[14] Jacobus E. van der Merwe, Sean Rooney, Ian Leslie, and Simon Crosby. The Tempest—a practical framework for network programmability. *IEEE Network Magazine*, 12(3):20–28, May 1998.

[15] Sean Rooney, Jacobus E. van der Merwe, Simon A. Crosby, and Ian M. Leslie. The Tempest: A framework for safe, resource-assured programmable networks. *IEEE Communications Magazine*, 36(10):42–53, October 1998.

[16] Richard Mortier, Rebecca Isaacs, and Keir Fraser. Switchlets and resource-assured MPLS networks. Technical Report No. 510. Cambridge University Computer Laboratory, UK, July 2000.

[17] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[18] Herbert Bos. *Elastic Network Control*. PhD thesis, Cambridge University Computer Laboratory, UK, August 1999. Available as Technical Report No. 483.

[19] James W. Stamos and David K. Gifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[20] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7), September 1996.

[21] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proceedings of IWAN'99*, pages 25–36, Berlin, July 1999. Springer-Verlag.

[22] Sean Rooney. Connection Closures: Adding application-defined behaviour to network connections. *Computer Communication Review*, 27(2):74–88, April 1997.

[23] A.A. Lazar, K.S. Lim, and F. Marconcini. Realizing a foundation for programmability of ATM networks with the binding architecture. *IEEE Journal on Selected Areas In Communications*, 14(7):1214–1227, September 1996.