Application-specific behaviour in distributed 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

Abstract— **An innovative control architecture for ATM is presented that allows users to reserve in advance arbitrarily complex connections. It further allows the advance reservation of partitions of a virtual network which are called netlets. The network control is elastic, i.e. active and extensible, in the sense that applications are able to load code into the control architecture to manipulate network resources at a very low level. In this way new application-specific operations can be implemented. Running code in the heart of control architectures makes such problems as access control and trust extremely important and a simple solution for them is discussed. Finally, it is possible to load code on a per switch port basis which is able to influence admission control decisions made for this port.**

Keywords— **active networks, distributed open control, reservations in advance, agents**

I. Introduction

Resource Management in networks, be they ATM networks or IP networks with signalling protocols such as RSVP, involves first and foremost reservation and allocation of resources to specific applications. Giving applications guarantees as to the availability of resources is necessary whenever Quality of Service (QoS) guarantees are needed. Resource Management and QoS guarantees have been a part of ATM from the very outset, and are now rapidly gaining ground in the IP world as well. In this paper, we focus on ATM but many of the issues are equally relevant in other network technologies. The paper is motivated by the recognition that there may be many policies pertaining to resource management, each of which may be suitable in specific application areas, but not in others.

A. Reservation in advance

An active research topic in the field of resource management is the problem of resource reservation in advance (both in IP [1], and in ATM [2], [3], [4]). Whether advance reservations are needed at all, depends on the scarcity and value of resources. If resources are scarce, there exists a class of applications which would benefit from advance reservations. The criterion is whether the value of the advance reservation outweighs the cost of making the reservation. This is analogous to reservations in other areas. For example, it is worth the cost to reserve a seat in a plane, but it makes no sense to do the same for a trip by subway. Again, there are many primitives that can be offered for advance reservation and allocation of resources (e.g. simple end-to-end, simple multi-cast or even complicated timeshared multiplexing multicasts as in [2]), each of which suits its own application area, but not necessarily any others. Only applications know which operations and what resource management policy suit them best.

Following [5], we define *control architecture* as the set of protocols, policies and algorithms used to control a network. Our goal was to build a distributed control architecture that implements commonly used operations efficiently, while allowing applications to extend this basic functionality. Following the terminology of [6], we call such a control architecture *elastic*. The control architecture should be flexible enough to support both *arbitrarily complex* advance reservation and immediate connection setup.

B. Addressed problems

First, the generic nature of high-level primitives prevents applications from exploiting application-specific knowledge. Consider figure 1. The nature of the application is such that the choice of which endpoint is source is governed by an application-specific policy. Ideally, we would like to leave all connections to and from the central switch S in place so all that is required is changing the switch connection in S. High-level end-to-end primitives are incapable of exploiting this.

Second, it should be possible to extend the control architecture with new operations. This allows for greater flexibility and has also proved to be extremely useful in prototyping. Third, it is possible to partition networks into virtual networks each of which can be controlled by its own control architecture [7]. This partitioning is rather heavy-weight. For example, each virtual network has to be policed to ensure

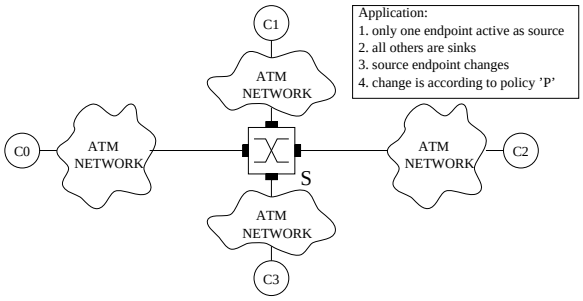

Fig. 1. Application-specific knowledge

that it doesn't use more than its allocated bandwidth. It would be useful to be able to repartition in a more light-weight manner, where policing and other policies are determined by applications. As far as we know, no such light-weight virtual network solutions have been described in the literature. Fourth, an adequate access control mechanism is needed, and related to this, there is need for establishing trust between clients that want to inject application-specific code into the control architecture and the control architecture. In other words, the problems of security and trust are very important. Finally, advance reservations may take arbitrary forms that may be difficult to express in high-level primitives. An example of such a reservation is given in figure 2. Such reservation policies are not supported in existing advance reservation schemes.

EXAMPLE OF AN ORGANISATION'S RESOURCE RESERVATION POLICY

1. Reserve a network N every day from 9am till 5pm
2. Except on Sundays in which case the reservation should be from 10am till 4pm
3. Except when that Sunday is May 1 in which case no network is needed
4. The amount of bandwidth to reserve on the switches in network N is B
5. If after 3 weeks the utilisation of N has never reached above 0.75B:
   -change the amount of bandwidth to reserve to: B = 0.9B
6. Repeat (5) every 3 weeks

Fig. 2. Example of arbitrary reservation request

C. Overview

We address all problems of section I-B. The research context will be described in section II. In section III we mention some basic operations of the control architecture. Light-weight virtual networks are introduced in section IV. Dynamically loadable code and its interaction with these networks is discussed in section V. section VI addresses the issues of access control and trust establishment. We show how arbitrarily complex reservations can be supported in section VII. Some performance figures are mentioned in section VIII. Section IX, discusses related work. Section X, finally, will summarise and draw conclusions. An implementation of the control architecture, called *Sandman*, has been achieved and is currently running on an ATM testbed. Implementation issues will be discussed throughout the text.

II. One size does not fit all

There is no such thing as a one-size-fits-all solution for control architectures. Many control architectures exist today already (e.g. Q.2931, RSVP, P-NNI, IP Switching, etc.) and it is not realistic to expect any of these to evolve into the ultimate control architecture that will cater to all our needs, present and future. Instead, we would like to enable clients to control their networks with the control architecture that suits their environment best.

For this purpose, previous work has allowed us to partition physical networks into virtual networks, each of which can be controlled by its own control architecture (so, effectively we have multiple control architectures controlling a portion of the same physical switch) as described in [7]. This is illustrated in figure 3, where a switch divider process partitions the resources on a switch into so-called *switchlets* and offers, for each of the switchlets, the exact same switch interface to the control architectures as found on the switch itself. However, the interfaces pertain to a switchlet and are therefore only able to manipulate a switchlet's resources. In the figure, three different control architectures are active, one of which is RSVP. It appears to each of the control architectures that they are controlling a real (albeit smaller) switch.

Switchlets are created and destroyed on behalf of an entity called the *virtual network builder* (or *netbuilder*), which is capable of creating and then combining a number of switchlets on different switches into a *virtual network*. Again, the virtual network is really a subpartition of the physical network. The virtual networks are completely independent in the sense that one control architecture cannot access the resources belonging to another control architecture and if connections misbehave in one virtual network, this has no effect on the other virtual networks.

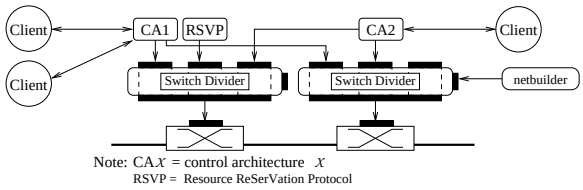

Fig. 3. Partitioning switches

In this light we introduce a control architecture called *Sandman*, which takes the idea of the infeasibility of a one-size-fits-all solution into the control architecture itself. It allows both immediate call setup and advance reservation and is extensible to accommodate any user-defined control policy. In this sense, we speak

about *open control*: control that is not dictated by any one standard, organisation or network operator. Even so, it is by no means intended to replace existing control architectures. Rather, it is expected to run alongside them as in figure 3. Sandman by its very nature is a distributed application. It consists of components that are spread out across a network and communicate using a distributed processing environment (DPE). The current implementation is built on a CORBA compliant DPE called DIMMA [8].

III. Basic operations

The Sandman supports a few commonly used operations that are expected to be sufficient for many applications. Each of these allows for reservation in advance, so that guarantees about the availability of the required resources at some time in the future can be given. The simplest and probably most common operation is the connection from source to sink for a particular time interval with particular characteristics (e.g. a peak rate of 1000 units/s). A client submits a request for such a connection to the Sandman using remote procedure calls (RPCs) over IIOP. If the admission control accepts the request, the client is guaranteed that the connection will be set up in that time interval. Traditional, immediate, connections leave out the interval in which case it defaults to $[now, \infty\rangle$. 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, are described in [2].

IV. Netlets

The basic operations are sufficiently expressive for a large number of applications, but not for all. Some applications simply want to make a reservation for a number of resources which are theirs to use as they please (without any connections imposed on them). Also, it has been suggested in [4] that resources in a (virtual) network be partitioned, so that immediate reservations are shielded from advance reservations (and *vice versa*). For this purpose the Sandman allows one to make advance reservations for what we have called *netlets*, which are small virtual networks in our larger virtual network.

Netlets consist of an arbitrary set of resources inside an encompassing virtual network. For example, for switch port resources we specify a netlet element which includes the switch name, the port number, the number of channels (this loosely translates to VCIs in ATM) and bandwidth. These netlet elements need not be adjacent as one netlet may consist of multiple unconnected sub-partitions (see figure 4).

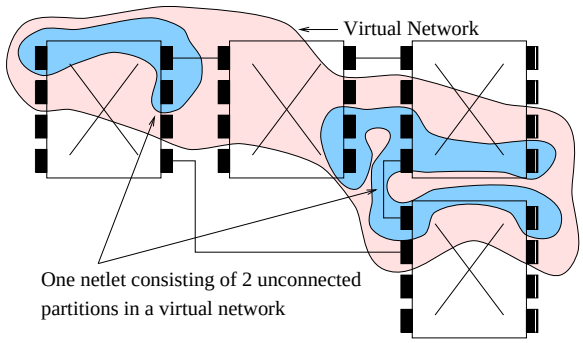

Fig. 4. Netlet in virtual network

A. Recursive repartitioning

At the start of the reservation interval, the netlet resources are allocated to the requesting client. The client can now set up connections, tear them down or, in short, do as it pleases with these resources. At the end of the interval, the Sandman automatically releases all resources belonging to the netlet. Netlets can be created recursively, i.e. 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 itself can be thought of as a netlet (the so-called *level-0 netlet*).

Netlets allow us to re-partition the capacity of a virtual network, so that we can, for instance, give an application (or group of applications) a certain amount of bandwidth on a number of ports, without creating a separate 'hard' virtual network for it. Applications can manage their own connections without requiring a new instantiation of a (possibly heavy-weight) control architecture. In other words, netlets are lightweight virtual networks[1]. As we shall see in section IV-B, netlets enable us to exploit application-specific knowledge that allows us to make certain operations or network policies *lighter*. To enable applications to use netlets, we implemented lightweight operations for immediate connection setup in netlets. We are now able to make an arbitrary number of partitions (that is more general than the partitioning mentioned in [4]) between, for example, immediate reservations and advance reservations. More advanced netlet-control will be discussed in section V-B.

B. Policing and propagation of damage

It is important that the virtual networks of section II are policed, because misbehaviour in one virtual network should not affect connections in any of the other networks. Given hard (in-band) virtual-network

[1] the relation between a netlet and a virtual network is similar to that between a thread and a process in operating systems

policing, however, we note that the partitioning of virtual networks into netlets may be logical, e.g. although a certain amount of resource has been reserved for a netlet (on the basis of which CAC decisions are taken), it may not be necessary to actually police the behaviour of the netlet. For example, we may assume (or know) that sources in the netlet will never exceed their allocated bandwidth (using application-specific knowledge). Or we may not implement policing because it is cheaper and faster not to do so.

Even if a particular connection misbehaves (e.g. uses up more than its allocated bandwidth), the problems will be limited to this virtual network only (since the virtual network itself is policed). Higher-level netlets therefore may be not policed at all, or policed via a looser kind of policing, where a process monitors netlets by periodically taking measurements. Misbehaving connections can then be reported (slightly *after the fact*) to the control architecture that subsequently decides on the appropriate action (e.g. tear down the offending connection).

For loose policing, we implemented so-called *traffic servers* which run close to (or even on) switches. Traffic servers can be instructed by external clients with the appropriate access rights, to measure periodically the traffic on certain connections and send notifications if the amount of traffic exceeds a certain threshold. Traffic servers are able to monitor netlets at any timescale that is thought appropriate.

V. Fine-grain control and loadable code

The resources of netlets as described in the previous paragraph are said to *belong* to specific applications. This means that these applications should be able to manipulate these resources in any way they want, enabling them to use application-specific knowledge about the behaviour of the connections and the traffic. Consider again figure 1. We need control at a finer level of granularity than end-to-end connections to exploit the knowledge about the application described in section I-B. A simple way of solving this is by exporting operations for low-level control of a netlet's resources to the outside world (e.g. the operation to create a connection across a single switch), which clients can call using RPC. The problem with this solution is that sending the low-level commands across the network is a slow operation and hampers performance if many such commands have to be executed (e.g. a setup across a large number of switches).

This is a problem that is inherent to distributed applications and any distributed system should take it into account. An elegant solution is to enable the application to pass its own application-specific control policy[2] into the control architecture and have it interact with very low-level control operations [5]. Note that this is different from what is commonly called *active networks* [9] in the sense that it keeps a clear distinction between control and datapath, while *active networks* are generally understood to interpret the packets on the datapath. Similar to active networks is that we have tried to standardise the computational model. The model used is called the *Sandbox* model, where the Sandbox represents the execution environment in which the DLA is executed. The Sandbox restricts the access of application-specific code to resources owned by the application. Every Sandbox contains a simple, extensible interface for DLAs that allows DLAs to export interfaces, communicate with peers, etc.

A. Issues

A primitive operation in the Sandman allows applications to load code into the network. This is illustrated in figure 5. The Sandman's connection manager (CM) is the central entity in a Sandman, which handles the communication with the physical switches. Clients communicate with the CM to have it do things on their behalf, e.g. set up connections on a switch, tear them down, load application-specific code, etc. The loadable code runs in the Sandbox, which resides in the same address space as the CM, so the communication overhead between them is minimal[3].

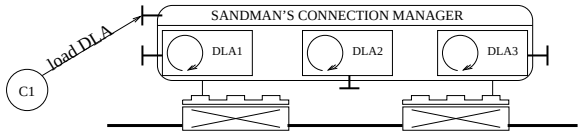

Fig. 5. Application code injected in the network

The operations that are made available to the application-specific code range from the normal operations that are available to normal applications (e.g. reserve a connection, reserve a netlet, etc.), to low-level operations and operations that gather information about the network.

The DLAs are self-scheduling. In other words, the application specifies the initial time that it wants the DLA to be executed and after that the code itself manages its wakeup time. To help it do this, the Sandman provides the DLA with a context (such as the current time) each time it executes it. Using loadable code, it is possible to extend the control architecture by implementing new operations which may be called by any

[2]We call such code a *dynamically loadable agent (DLA)*.

[3]Implementations exist for Tcl and partially for Java and Python

client in the outside world. In other words, it is possible to *export* some of the application-specific code for general use.

B. Associating netlets with behaviour

We can now associate application-specific behaviour with particular (sets of) netlets to create what is defined in [5] as a connection closure: a set of resources together with a policy that manipulates these resources. In this way, applications can implement any resource allocation policy on the network that suits their environment. The mechanism proposed here is similar to connection closures, but more generic in the sense that the loadable code is able to create new netlets (if need be recursively), which it can control itself as separate light-weight networks, or associate with a new DLA to effectively create a new closure.

VI. Security and access control

Running foreign code in the heart of the Sandman introduces risks that range from code affecting other applications or stealing sensitive information, to code running computationally intensive programs that take up a lot of CPU time. The first problem could be handled by careful shielding between DLAs and the rest of the Sandman while the second can be dealt with if we use an operating system such as Nemesis [10] which allows us to control the amount of resources (such as CPU time) that a specific application is able to use. Shielding between two DLAs can be achieved by using different Sandboxes. Provided these Sandboxes' access to resources is properly bound, a DLA in one Sandbox cannot corrupt data in either the underlying system or in the other Sandbox. The details of this solution are beyond the scope of this paper.

A. Access control

Access control to the Sandman's Sandbox operations is implemented using primitive *capabilities*. As discussed in section V-A, DLAs may implement some new functionality and export a reference to this functionality to the outside world (e.g. via some trading mechanism). External clients can then call this new operation, which extends or even overrides the existing functionality, as long as they present the appropriate capability. Note that the entire Sandbox functionality is protected by such capabilities, not only the new operations created by DLAs. Who creates these capabilities, or how, is beyond the scope of this discussion.

B. Trust

There is another issue related to security which deserves our attention. In systems that allow foreign code to be loaded in the local execution environment, the focus of safety policies is often on protecting the environment from malicious code. However, a more subtle problem arises when clients are not sure whether or not the Sandbox, in which their DLA will run, can be trusted. Malevolent Sandboxes may corrupt or steal sensitive information from their DLAs. This may be a common problem in environments when DLAs may be mobile (as is the case in the Sandman).

The problem of untrusted Sandboxes is solved by trying to establish a *trust relationship* via a trusted third party, known as the *trust* server using a special method in the Sandbox interface called `establishTrust` (see figure 6). The primitive mechanism relies on public/private key cryptography and certification. Public/private key encryption is symmetrical in the sense that a message encrypted with a principal's public key can be decrypted (only) with the private key and, *vice versa*, a message encrypted with the private key can be decrypted by everyone who owns the public key (but could only have been encrypted by the principal owning the private key, which makes it a useful mechanism to sign messages[4]). All messages being exchanged are encrypted. At start of day, a trusted *certification issuer*, is responsible for issuing unique, unforgeable certificates to each *bona fide* implementor of Sandboxes. The certificate, or *secret*, is embedded in the implementor's Sandboxes, inaccessible to foreign code.

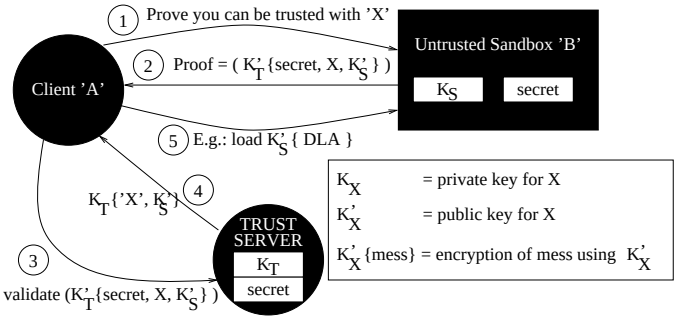

Fig. 6. Establishing trust via the trust server

A client A that wants to run a security-sensitive DLA in a (so far) untrusted Sandbox B requests the Sandbox to prove that it can be trusted. The proof should include a specific message X specified by the client. X could be a string concatenated with a monotonically increasing sequence number, or in a more general case, a *nonce*. This is illustrated in figure 6 by **(1)**.

At that point, a *bona fide* Sandbox constructs a proof that consists of the Sandbox's secret, its pub-

[4]The disadvantage is that it is fairly heavy-weight.

lic key, a signature identifying the implementor and the nonce X specified by the client. It encrypts the proof using the trust server's public key and returns it to client A **(2)**. Next, client A requests a (known and trusted) trust server to validate the proof, as it is not able to read the proof itself **(3)**. The trust server knows about B's certificate. It decodes the message using its private key and finds the original secret corresponding to the implementor's signature. It then checks whether the result equals the secret sent by the Sandbox. If there is a match, it returns an acknowledgement which includes the nonce X and B's public key **(4)**. The response is encrypted with the trust server's private key, which ensures that it cannot be forged by some other entity. The client decrypts the response and checks whether the nonce matches the one it expected to receive. If so, the trustworthiness of Sandbox B is established. It can send its security-sensitive DLA to the now trusted Sandbox encrypting it with B's public key **(5)**.

VII. Applications and Admission Control

Using loadable code we are able to extend the functionality of the Sandman with application-specific policies. We are able to repartition the resources in the virtual networks into netlets or light-weight virtual networks, which can be associated with application-specific policy if needed. The remaining problem concerns arbitrary advance reservations, as exemplified by figure 2. Observe that simple reservations (such as a reservation for a specific netlet tomorrow from 9am to 5PM) which are traditionally used for advance reservations are not sufficiently expressive to deal with this simple example. Due to the long time scale it is not possible to make a separate reservation for every single day we need a netlet because the number of entries in the reservation schedules would explode. Again, an elegant solution is to use loadable code, although in a manner that goes beyond the connection-closure like behaviour of section V.

A. Admission Control

In the following, we will 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). Figure 7 illustrates the separation between the two types of behaviour.

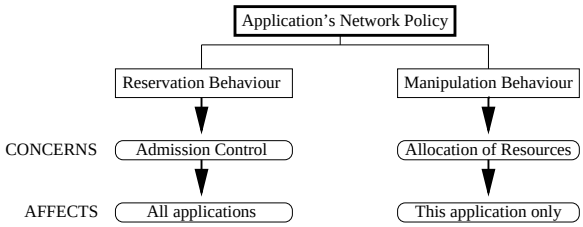

Fig. 7. Reservation behaviour vs. manipulation behaviour

An important difference between the two is that manipulation behaviour is not supposed to affect any other applications (the effects are limited to its owner application) whereas reservation behaviour is. For example, setting up a connection for application 1 should have no effects on application 2 as it does not change the state of the resources application 2 expects to control. Reserving resources on the other hand is something that may affect all other applications in the system as it prevents them from reserving and using these resources during the interval specified. It is important to realise that resource reservation behaviour is essentially an input to the call admission control (CAC), whereas manipulation behaviour is not.

B. Application-specific reservation behaviour

We have seen that DLAs allow freedom in the resource manipulation behaviour. Applications can load up a DLA and have it set up connections and manipulate resources on their behalf. The reservation behaviour, however is still fixed. Clients are only able to reserve connections, or netlets, for specific time intervals. We call this type of reservation behaviour, where all bookings are entered in some reservation schedule, *static*. We think this is the case in all advance-reservation control architectures to date and will always be the case if high-level primitives for reservations are used, whatever the primitives may be.

We will now show that we can open up the reservation behaviour as well. This is done by allowing applications to load code into the Sandman that is capable of influencing the CAC decision. We call this type of reservation behaviour *dynamic*. Recognising that it is essentially a CAC problem that we are addressing, we call instances of such code *CAC DLAs*. Sandman offers an interface with an operation that is capable of installing CAC DLAs for a particular port of a switch[5]. Figure 8 shows the two types of loadable

[5]The granularity of switch port is an implementation detail.

code running in the heart of the Sandman.

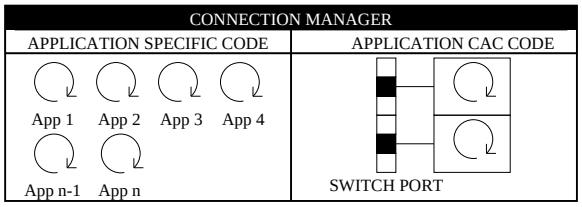

Fig. 8. The two types of loadable code running in the CM

C. Temporary reservations at CAC time

The way in which the CAC DLAs can influence the CAC decision is conceptually as follows. Whenever the Sandman needs to make a CAC decision for a switch port P (i.e. it tries to make a reservation R for a certain amount of bandwidth for a particular time interval $[T_{start}, T_{end}]$), it checks whether any CAC DLAs were installed for P. If so, it will execute this code. The CAC DLA receives from the Sandman the time interval $[T_{start}, T_{end}]$ corresponding to reservation R. It then makes *temporary reservations* according to its own reservation policy for this time interval. After the DLA has made all the reservations it wants to make for this interval, it simply returns. The CAC algorithm will now make its decision based on all reservations currently in the schedules, including the temporary ones. The temporary reservations disappear automatically when the CAC procedure returns.

As an example, consider a contrived application A_1 that wants to reserve bandwidth proportional to what day of the month it is. So, if it is the first of that month, the application reserves B bandwidth, next day it reserves $2B$, the day after that $3B$ and so on until it wraps back to B, when the month changes. Also assume that the application's request to install a DLA corresponding to this resource reservation behaviour on a certain switch port was accepted. Suppose furthermore that the total capacity of that switchport is B_{tot}, that the amount of bandwidth reserved statically (in reservation schedules) is B_{resv} and that the available bandwidth at some point in time is B_{avail}. Now, a new application A_2 submits a request to reserve B_{req} bandwidth on this port from the morning of July the 5th until the night of July the 6th. What happens is the following:

1. The CAC procedure discovers that a DLA exists for this port and executes it.
2. The CAC DLA makes temporary reservations for $6B$ bandwidth for the interval and returns control to the default CAC procedure.

It could be changed to providing loadable code per switch, or per small number of channels on a particular port, if so desired.

3. The default CAC procedure now simply checks its schedules and accepts the new reservation request if at all times during the two days of interest $B_{avail} = B_{tot} - Bresv \geq B_{req}$, and rejects it otherwise.
4. The CAC result is returned (the temporary reservations disappear automatically).

Note that the CAC DLA is called each time a CAC decision is made for this port (in this virtual network). This will add some overhead to the CAC procedure. We therefore suggest that in a commercial situation applications that require this kind of flexibility in their resource reservation behaviour are charged a premium rate for their reservations, while applications that don't need this much flexibility are charged less.

D. Combining application-specific CAC programs

As explained before, the CAC DLAs are different in nature from the loadable code in section V in that it affects the CAC for all applications. The CAC DLA describes resource reservation behaviour which cannot be specified using higher-level operations. This introduces the following interesting problem: how do we decide whether or not the resource reservation behaviour as specified in the CAC DLA of one application will never ever clash with that of another? In other words, how do we decide that two loadable CAC DLAs are mutually *feasible*?

Depending on how general one allows CAC DLAs to be, this may be a problem that needs to be decided off-line. In practice, instead of specifying DLAs in Turing-complete languages which take up an arbitrary amount of CPU time, one probably wants to be more restrictive. For example, DLAs may be restricted to a limited execution time (e.g. ≤ 5 ms), and/or it may be restricted by way of restrictions on the programming language. Instead of a general-purpose language, a programming language could be used that is easy to check at run time. As a restrictive but still powerful example, consider the case where the DLA can only consist of a limited number of statements of the form:

```
if <expression> then <temporary_reservation_list>
```

where `<expression>` and the amount of resource to reserve are simple functions of the time. In other words, the DLA is written as a list of function descriptions $f_i(t)$ that determine what and how much needs to be reserved. Suppose that N CAC DLAs have already been loaded for a resource with resource capacity C and that a client would like to install another one. The feasibility check now consists of determining whether the following holds:

$$\max_t \left\{ \sum_{i=1}^{N+1} \sum_j f_{i,j}(t) \right\} \leq C \qquad (1)$$

where $f_{i,j}$ denotes the j^{th} function description of the i^{th} CAC DLA.

VIII. Performance

Due to limited space, we won't discuss many results. However, to indicate that exercising network control from DLAs consisting of interpreted code need not impede performance, we compare two scenario's. In scenario A, 1000 connections are set up and torn down across a FORE ASX-1000 switch, from the Sandman *directly* (without DLAs). In scenario B, the same is done, but now the setups and teardowns are performed by a DLA. In both cases, the switch divider runs off-switch on a Sun UltraSparc and the client of the switch divider runs on yet another UltraSparc. In scenario A, the average time for a setup + teardown was 29.8 ms. In scenario B, the average duration was 30.4 ms. The difference is minimal. Moreover, the figures compare favourably with other figures for connection setup, both on-switch and off-switch [11].

IX. Related work

For advance reservations, [3] is interesting as it discusses the requirements of the clients of an advance reservation service, and proposes a distributed design for such a service. The network is partitioned to separate *immediate* setup and *advance reservation*. A similar partitioning is made in [4]. The admission control in [1] combines advance reservations with measurement based admission control. It is aimed at bounding delay in the internet. Advance reservations and QoS negotiation is the topic of [12].

In network programmability, we distinguish between solutions that maintain a clear separation between control and data path and solutions that don't. An example of the latter is what has come to be known as *Active Networks* [9]. Active Networks are packet-switched networks where each packet may carry executable code. The execution of the packets in the data path is strongly related to the speed with which these packets can travel through the network. In Intelligent Networks (IN) [13], basic calls are separated from IN-based calls. For example, dialling an 0-800 number temporarily suspends the call-processing while the corresponding application-specific *service logic* is found and executed. In [6] a solution to network management using *delegated agents* is proposed, where the agents can be dispatched using a delegation protocol to an executing elastic server. Connection closures [5], finally, are related to first-level netlets with an application's resource manipulation behaviour. There is, no temporal aspect to connection closures and reservation behaviour is fixed. Also, it is not possible to extend the control architecture with new "public" operations.

X. Conclusions

We discussed an elastic control architecture which enables applications to inject their own policies into the network. The control architecture supports immediate and advance reservations. Furthermore, it allows fine-grain control over the resources and allows the default functionality to be extended. The resources in the network can be recursively repartitioned at a fine granularity using netlets. A netlet can be associated with application-specific code. Attention was given to access control and trust establishment. Finally, it is possible to load arbitrary resource-reservation behaviour into the network. Elastic network control allows applications to exploit application-specific knowledge that could never be captured by a fixed set of high-level primitives.

References

[1] M. Degermark, T. Kohler, S. Pink, and O. Schelen, "Advance reservations for predictive service," in *NOSSDAV*, Lecture Notes in Computer Science, (Durham, New Hampshire), pp. 3–14, Springer, Apr. 1995.

[2] H. Bos, "Efficient reservations in open atm network control using online measurements," *International Journal of Communication Systems*, vol. 11, pp. 247–258, Aug. 1998.

[3] D. Ferrari, A. Gupta, and G. Ventre, "Distributed advance reservations of real-time connections," *Multimedia Systems*, vol. 5, no. 3, pp. 187–198, 1997.

[4] A. Schill, S. Kuehn, and F. Breiter, "Resource reservation in advance in heterogeneous networks with partial ATM infrastructures," in *Proceedings of INFOCOM'97*, Apr. 1997.

[5] S. Rooney, *The Structure of Open ATM Control Architectures*. PhD thesis, University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., Feb. 1998.

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

[7] 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.

[8] G. Li, "Dimma Nucleus Design," Technical Report 1553.00.05, APM, Cambridge, U.K., Oct. 1995.

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

[10] 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.

[11] 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.

[12] A. Hafid, G. von Bochmann, and R. Dssouli, "Quality of service negotiation with present and future reservations: A detailed study," *Computer Networks and ISDN Systems*, vol. 30, May 1998.

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