Efficient Reservations in Open ATM Network Control using Online Measurements

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— **We propose an innovative control architecture for ATM that allows users to reserve in advance complex connection patterns in the network. It also allows the advance reservation of partitions of a virtual network which are called virtual netlets. The control architecture offers control over resources via peak-rate based schedules but alleviates the potential over-conservative nature of the resource allocation by coupling it with real-time measurement of actual use of the resources.**

Keywords— **ATM admission control, effective bandwidth, measurements, reservations**

I. Introduction

Present-day technology allows users to access continuous media (CM) data as well as traditional types of data simultaneously across a network. Problems arise when resources (e.g. file servers or network links) are overloaded and either degrade the system as a whole or cause new requests that want to share the resources to be rejected.

Size, load balancing and natural distribution often cause sources to be distributed across networks. For example, HDTV video requires 17.5 Megabytes per second of video (CBR encoded [7]), a few hours of which will not fit completely on most storage devices and may well be segmented and distributed. Load-balancing may be an even more important reason for distributing data sources. In [3] a policy is proposed that balances load by chopping up CM files in segments, which are dynamically distributed over the file servers, depending on the load. Furthermore, audio, video and other types of data may be stored on specialised servers for efficiency [11]. Finally, many sources are distributed by nature. Cameras and microphones for example, are attached to workstations or, in the case of security cameras, distributed over an organisation's site.

Source distribution becomes a problem if acceptance of a call to one source depends on the acceptance of calls to a set of other sources. We call this *temporal connection dependency.* A connection is temporally dependent on other connections if its establishment is only meaningful in the context of the (possibly future) establishment of a number of other connections.

For example, if a video file is distributed over 4 nodes (see figure 1), the playback of the entire video requires that segment 1 is played first, immediately followed by segment 2, etc. It is not acceptable that the first 3 connections are accepted while the last one is rejected (which means that the client is not able to watch the end of the movie). Note that it is also not acceptable to set up all 4 connections in advance for the entire duration of the video, since this would be a needless waste of bandwidth (all 4 connections would be idle for 75% of the time). Instead, it is required that the Admission Control guarantees that if the connection to source 1 in a sequence is accepted, the connections to all subsequent sources are also accepted at the appropriate times. So connections 1 to 4 in figure 1 are said to be temporally interdependent. These sorts of guarantees are required in all systems with temporal connection dependencies. A 'live' example might be a video conference where a number of speakers have been allocated speaking time in advance.

As to whether advance reservations are needed at all, we follow [4]: this depends on the future scarcity of resources. Where resources are plentiful, not even immediate reservations are necessary. If resources are scarce enough to justify immediate reservations, advance reservations make sense as well. Note that the only alternative to reservations is over-provisioning of resources. Also, looking at the past few decades, one can observe that over-provisioning of resources (be it for memory, storage capacity or indeed network bandwidth) does not have a very good record: whatever the capacity of the resource, people have found ways to use it all.

We define a *control architecture* (CA) as the set of protocols, policies and algorithms to control a network [15]. This paper introduces a control architecture capable of meeting the requirements of temporally dependent connections. Operations will be introduced that allow one to make reservations of an almost arbitrarily complex nature. We will show that by using these operations, partitioning the resources to distinguish between immediate and reserved connections is trivial. Some thought will be given to the sort of state that is introduced in the network and the way in which faults are handled. Finally, we introduce an admission control policy which prevents over-conservativeness in resource allocation resulting from peak-based reservation.

Section II contains background and related work. In section III we describe our general attitude towards control of ATM networks. Section IV discusses the components of the CA. Section V describes the call admission control. Some thought to the nature of the state and failures is given in section VI, while results are presented in section VII and section VIII draws conclusions.

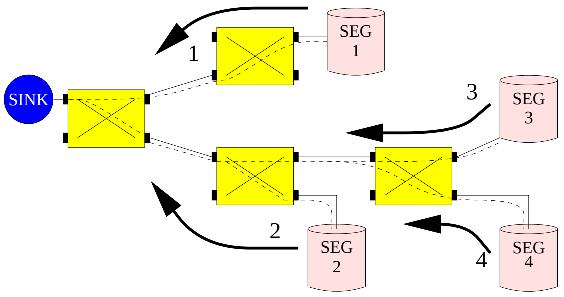

Fig. 1. One CM file consisting of 4 segments

II. Background

Much research is devoted to resource reservation on the datapath. In the Internet for example, RSVP is a signalling protocol that offers receiver-initiated reservation (which suggests that the IP and ATM worlds are converging). In ATM meanwhile, a wide variety of Call Admission Control (CAC) algorithms for resource management has been proposed. Some of these use very clever estimation of the total bandwidth that is required for n connections of which some properties (e.g. peak rate, average rate, burst length) are known. Gaussian Approximation and Equivalent Capacity[5] are examples of these algorithms.

Most of these algorithms suffer from relying on a (static) model of the traffic, while often it is impossible to accurately characterise sources. An approach that is frequently taken in existing policies is the division of calls in a small set of Quality of Service (QoS) classes. This technique is problematic also, since one can never define classes that fit all possible types of traffic (including those of future services). Finally, most existing CAC algorithms differ from the policy proposed here in that they deal with whether a call can be accepted *now* rather than at some time in the future, which is needed for the guarantees mentioned above.

Recently there has emerged a promising trend to use on-line measurements to overcome some of these problems, e.g. [2] and [8]. In [8] we find the mathematics for a CAC policy similar to the one described here. Several schemes are described and a method is discussed for assigning priorities to traffic classes. No detailed source characterisation is required.

Independent research [4] (with effective bandwidth estimations based on [6]) deals with problems similar to those addressed in this paper for IP based networks. It offers advance reservations and measurements-based admission control. It is quite different in its scope and objectives, however, as it focuses on bounding delay, while we try to manage bandwidth in a more general way. True to its objectives, [4] allows only simple reservations, whereas we deal also with the way in which connections are related. For example, we can trivially repartion whole networks to provide guarantees and reservations to related connections, which also gives us the possibility of bounding the amount of bandwidth that is being used by applications without restricting the number or nature of their connections. Novel connection types that implement ATM style multipoint connections allow us to make advance reservations for most types of applications, while the possibility of making advance reservations for arbitrary virtual networks caters to the needs of the remaining applications for something even more application. Other differences include the network technology and equivalent bandwidth estimation algorithm (which is very simple and crude in [6]).

In [16] it is argued that it may be useful to distinguish between immediate reservations and advance reservations. We have implemented such a partitioning in a more general way, so that, for example, there can be an arbitrary number of partitions for immediate reservations.

Related work on control of ATM networks is carried out in the OPENET [1] and the Xbind projects [9]. Xbind describes a framework for multimedia services on ATM and allows the binding of networking resources to create distributed services. Unfortunately, the CAC relies on a static model of the traffic.

Finally, we mention that the control architecture builds on previous work in the DCAN project [14].

III. *The* Control Architecture doesn't exist

It is important to understand the context of this research. In particular, it is a fundamental belief of this work is that there is no such thing as a one-size-fits-all solution for control architectures. We observe that many control architectures are being used today (e.g. Q.2931, SPANS, P-NNI, IP Switching, etc.) and they all serve their purposes, but it is not realistic to expect any of these to evolve into *The* Control Architecture that will cater to all our needs, present and future.

As pointed out in [15], this is not only infeasible, it is also undesirable, due to the inherently bloated nature of such a control architecture, which would contain a lot of functionality that one would not need on a day to day basis. Instead, we would like to enable users to control their own networks with their own favourite control architecture that suits their environment best. For this purpose, previous work done in the Computer Laboratory [12] 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). In the most extreme case, where a control architecture exploits knowledge about the application domain, we speak about service specific control architectures [13]. Interoperability between control architectures A and B can be achieved by implementing the interfaces of A in B, or vice versa.

In this light we introduce a novel control architecture which is service specific in the sense that it is very good in application areas where advance reservation of resources is required. We speak about devolved control, since almost all control is moved out of the switch into a general purpose workstation, facilitating development and deployment of new services and allowing us to control very dumb devices.

IV. Control architecture

Guarantees about availability of resources require the control of all relevant resources in the system. We therefore associate a Local Resource Manager (LRM) with a small set of resources, e.g. those on its own host[1] and some dumb devices. The LRM keeps reservations for these resources in allocation schedules. The schedules make it possible to reserve bandwidth of a shared resource in advance for a specific time interval.

The control architecture (CA) controls the ATM network. This includes setting up and tearing down connections and managing the resources in the network. For example, if a client C wants to watch the video file of figure 1, the CA has to set up a connection from server 1 to C during time interval $[t_0,\ t_0 + length(seg_1)]$ and from file server 2 to C during $[t_0 + length(seg_1),\ t_0 + length(seg_1) + length(seg_2)]$, etc. All these calls require some resources (e.g. bandwidth) for which we need a resource management policy. We call such a set of connections a *session*. The CA enables one to reserve these connections for a time interval in the future using a simple interface. If all bookings that make up a session succeed, the client can be sure that in the absence of failures it will be able to have all the required connections at the appropriate times. In a session, the architecture reuses existing connections as much as possible, employing *multi-source* connections that accept data from a number of sources in sequence[2].

A. Multi-point connections

The CA employs a new type of connection to connect multiple sources to one sink in sequence. These are not separate connections. Instead, there is one connection that is *time-shared* by several sources. It behaves like a rattlesnake, with its head at the sink and its tail at one of the sources: only the tail moves when the source changes, the rest of its body remains unchanged. The first part of the connection is not even aware of the hand-off of the sources. It just listens on the same VC, the connection is never torn down until all sources have finished.

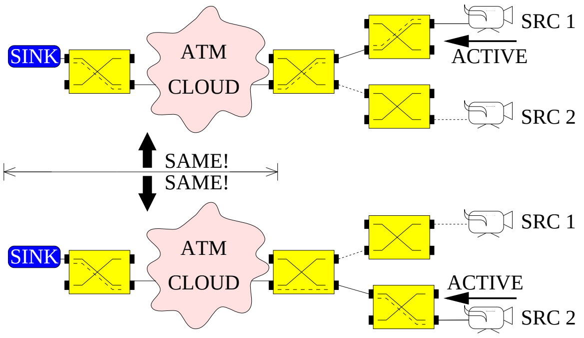

Fig. 2. Rattlesnake: only the source moves

[1]CPU, disk, network adapter, etc.

[2]This is quite different from such multi-point solutions as in OPENET [1], where multiple sources can send on a distribution tree so that a special adaptation layer is needed for packets larger than one cell.

For example, in figure 2 a client (sink) requests video-conference access to two cameras in sequence. The cameras are connected to 2 switches that connect to the same child switch. The paths from cameras to sink have an arbitrary number of switches in common (indicated by the clouds). At the top the connection is from camera 1 to the sink, while at the bottom we see that the camera has changed from 1 to 2. For the connection, however, hardly anything has changed. Only in the very last switch connections the change has been made from camera 1 to camera 2.

The first advantage of such a connection is that we don't have to go through the time-consuming process of setting up the entire connection from source to sink each time the source moves.We also do not waste resources for example by having multiple connections to the same sink in parallel in order to meet the requirements for temporal dependency between connections. And a third advantage is that the sink need not be aware of the handoff at all. The rattle-snake becomes a multi-headed dragon if there is more than one active sink (see section IV-C.3).

B. Components of the Control Architecture

We describe the various entities in the CA and their interactions (see also [14]). The components of the CA are built on a Corba implementation called DIMMA [10]. In the CA, we have the following entities (see also figure 3): *Host Manager*, *Local Trader*, *Connection Manager* and *Global Trader*. The Host Manager provides an interface to the CA. It also implements the functionality of the Local Resource Manager. Generally, there will be a Local Trader and a Host Manager associated with each host that wants to communicate (possibly running on the same machine).

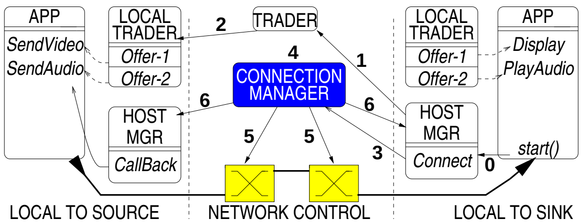

Fig. 3. Interactions between the various components

A typical interaction between these entities is[3]:

i. Before we start, the source exports a service offer, which is registered with its Local Trader, e.g. an offer to send video. The Trader also registers a callback function with this offer, which is the operation (e.g. "send") that will be executed when a client binds to the offer. Likewise, the sink registers an offer with its Local Trader, e.g. an offer to display video that is received from the network.
ii. When a client wants to access this service, it first tries to find an offer for it using its Local Trader, and if needed, the Global Trader, which in turn will talk to the other Local Traders until an offer has been obtained (1 and 2 in figure 3).
iii. It also obtains a handle on its local sink offer.
iv. The client, via the Host Manager, requests the Connection Manager to connect this source offer to this sink offer for this time interval (3).
v. The Connection Manager tries to reserve the appropriate resources on the data-path and if this completes successfully, it sends back an acknowledgment to the Host Manager.
vi. The request then sleeps (4) until t_{start}, at which point it sets up the connection (5).
vii. If successful, the Connection Manager tells the Host Managers on both sides to kick the callback operations (6): the source is told to send video and the sink is told to receive and display frames.

[3]described is a connection setup between source and sink program; this is by no means the only mode of operation of the CA, but here it is the most relevant one.

C. Interfaces

The CA offers an interface that allows users to do a wide variety of things. We will discuss only the small subset of the CA's operations that is relevant for systems with temporal connection dependencies.

C.1 Point-to-point connections

Everything is *timed* in the CA. Instead of setting up a connection between A and B, the user requests the CA to set up a connection from A to B, starting at time t_{start} and ending at time t_{end}. In this scheme, a traditional connection simply has interval $[now, \infty\rangle$. The operation for setting up a timed connection with a specific peak rate from source offer to sink offer is:

```
connectSrcToSink (Offer src, Offer sink, Bool pflag,
                  Bandwidth peak, Time t1, Time t2);
```

here `pflag` indicates whether connections should be made persistent, i.e. written to stable storage.

C.2 Connection sequences

A more advanced way of accessing the CA introduces the concept of *connection sequence*, which is a sink offer together with a list of source offers and times. Each item i in the list contains a source offer as well as an end time $t_{end}(i)$ corresponding to this source. For each item i in the list, the control architecture connects source_offer$_i$ to the sink offer until the time is $t_{end}(i)$, at which point it connects the sink to source_offer$_{i+1}$. Moreover, on the handoff, the CA reuses as much of the previous connection as possible, so we really have a single connection which is time-shared by multiple sources.

The result is a session with a single sink (receiving data on an unchanging vpi/vci pair) and a sequence of sources which follow one another seamlessly. Observe that the establishment of a connection sequence implements the rattle-snake connections described in (IV-A). This is the ATM equivalent of a (restricted) multipoint-to-point connection which prevents multiple sources from sending on the same VC simultaneously, so that there will be no interleaving of a client's AAL5 frames (with all problems associated with that). The operation signature is as follows:

```
connectSequence (List<OfferAtTime> reqsForSrcOffers,
                 Offer aSinkOffer, Time start);
```

where `OfferAtTime` is a type that corresponds to an item in the list. An element of this type contains a source service-offer reference, the peak rate for this source, as well as an end time, as follows:

```
class OfferAtTime { Offer src; Bandwidth p, Time t;}
```

The potential speedup in connection setup time on a source handoff (i.e. source S_i is replaced by source S_{i+1}) is simply the ratio of the number of switch connections that have to be set up:

i. let L_1 be the number of links from S_{i+1} to the first common node (FCN), i.e. the first switch shared by both the path from S_i and the path from S_{i+1};
ii. let L_2 be the total number of links from S_{i+1} to the sink;
iii. then the speedup ratio is: $\frac{L_2-1}{L_1} \times 100\%$.

Note that no time is lost tearing down the old connection. We simply set up the new connection to the FCN and, just before we set up the switch connection for the new path in the FCN, we remove the switch connection for the old path. This effectively disconnects the old source. We then set up the new connection and notify the new source via its callback function. Only after the new connection is active do we remove the dangling connection from the old source (the garbage collecting phase).

C.3 Connection patterns

The final step for connections is to extend the functionality of the connection sequence (which provides multiple sources) by adding multiple sinks to it. We call this a *connection pattern*. It allows users to specify a sequence of sources with corresponding non-overlapping intervals and associate with these a set of sinks (figure 4). The sinks are also tied to specific intervals but these *are allowed* to overlap (and may span several sources). This is the ATM equivalent of a (restricted) multipoint-to-multipoint connection.

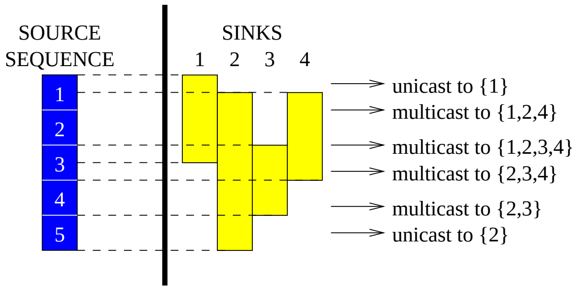

Fig. 4. Connection pattern request

Overlapping sink intervals indicate multi-casts: the data from the active source will be sent to all sinks that overlap with the source interval. In the CA, multiple sinks connect to an active source that already has a sink attached to it by way of a `join` operation. The `join` operation is called from within the control architecture and transparent to the application (it simply binds to the source offer)[4] The connection-pattern operation is as follows:

```
connectPattern(List<OfferInInterval> sources,
               List<OfferInInterval> sinks, Bool pFlag);
```

where `OfferInInterval` is similar to `OfferAtTime`, except that an interval instead of a single time is associated with the service offer:

```
class OfferInInterval { Offer anOffer; Bandwidth peak,
                        Time start;    Time end; }
```

Replacing sources in multicast trees

Observe that this last operation introduces an interesting problem regarding the activation of a new source. In figure 5, SRC1 acts as the source for a multicast tree. At some point SRC2 has to take over from SRC1. Instead of setting up a whole new tree from SRC2 to all sinks (which yields an *optimal* tree but suffers in performance because much of the existing tree *could* be reused), we adopted the following solution. We find the first node that the multicast tree for SRC2 would have in common with the old multicast tree, i.e. the first common node (FCN). We can simply reuse the tree underneath it. From

[4]the establishment of connection patterns is obviously contingent on the ability of the switch to support the corresponding multicast connections

the FCN we then establish a reverse connection to the root (more precisely: to the first node containing a multicast connection, counting from the root) and reuse all dangling subtrees along the way (figure 6)[5].

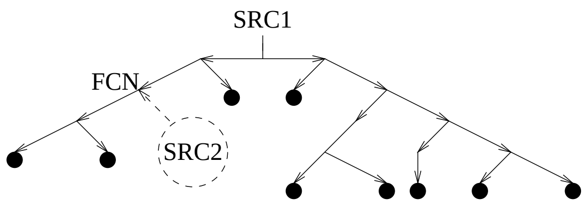

Fig. 5. A new source takes over

```
up = down = FCN = oldTree.firstCommonNode(newSource);
// if a subTree of the old tree exists at node FCN
if (oldTree.existsSubTreeAt (FCN)) {
   tree_t branch = oldTree.unconnectSubTreeAt (FCN);
   newTree.appendSubTreeAt (branch, FCN);
}
while (down != root) {
   up.goUp();
   // now reverse the direction of the connection
   newTree.createConnection (down, up);
   oldTree.releaseConnection (up, down);
   // if a subTree of the old tree exists at node up
   if (oldTree.existsSubTreeAt (up)) {
      tree_t branch = oldTree.unconnectSubTreeAt (up);
      newTree.appendSubTreeAt (branch, up);
    }
    down = up;
}
```

Fig. 6. Source replacement algorithm

C.4 Repartitioning virtual networks

Advance reservation of connection patterns provides users with very powerful and versatile reservation capabilities with explicit support for temporal connection dependency. This may not be enough for some applications, however. These applications may simply want to make a reservation for a number of resources which are theirs to use as they please (i.e. without any connection pattern imposed on them). Also, it has been suggested in [16] to partition resources in the (virtual) network, so that for example immediate reservations are shielded from advance reservations (and vice versa). For this purpose the control architecture includes the possibility to make advance reservations for virtual netlets, which are small virtual networks in our larger virtual network[6].

Virtual netlets consist of (a share of) an arbitrary number of resources inside the virtual networks. These resources need not be adjacent as one netlet (which by its nature partitions a virtual network) may itself consist of multiple unconnected sub-partitions (see figure 7). Also note that the partitioning 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). Now, even if they *do* misbehave, the problems will be limited to our own virtual network and not propagate to the outside world (assuming that there *is* a policing mechanism at the virtual network level). On the other hand, if we do wish to have a certain netlet policed, we can request the CA to do so.

At the start of the reservation interval, the resources are allocated to the client application which requested them (or some other application, in case of third party setup). The client can now set up connections, tear them down or, in short, do as it pleases with these resources. The CA guarantees that the client gets the capacity that it reserved on these resources. At the end of the interval, the CA automatically tears down all connections belonging to the netlet and releases the corresponding resources.

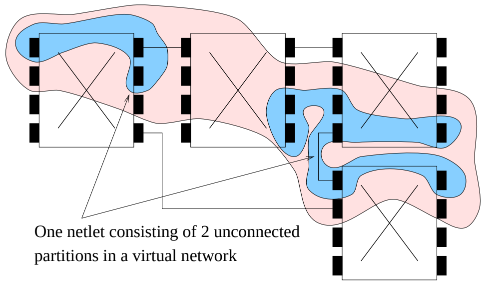

Fig. 7. Netlet in virtual network

So far, the only resource that is reserved for the netlets is bandwidth on switch ports. The operation for advance reservation of virtual netlets is as follows:

```
reserveNetwork (int32 &netId, List<vnetElem> *netlet,
                Time t_start, Time t_end);
```

where the `netId` (the netlet identifier) that is returned will have to be specified for each subsequent access to the netlet (e.g. to set up a connection in the netlet). A `vnetElem` uniquely identifies an amount of bandwidth on a specific switch port (with an indication whether it will be used as a source port or as a sink port) as follows:

```
class vnetElem { char* switchname;   int32 port;
                 PortDirection dir;  Bandwidth bw; }
```

Netlets allow us to re-partition the capacity of our virtual network, so that we can, for instance, give

[5]Note that calculating the speedup ratio is not straightforward as it depends on the algorithm to setup multicast trees and the way in which sinks are distributed in the old multicast trees, upstream and downstream from the new source.

[6]note that netlets differ from the operations of the previous sections, in that no connections are set up in them

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 (which would be the alternative). It can then manage its own connections without requiring a new instantiation of a possibly heavy-weight control architecture. In other words, netlets are light-weight virtual networks[7]. Communication between netlet and virtual network operations is fast as they execute in the same address space.

Also, in our control architecture we may want to implement functionality that is not (and should not be) provided by the normal virtual-network builder. For example we may want to enable netlets to change dynamically at very small time intervals, making calls to the virtual-network builder expensive. Or we may want to try out experimental policies to transfer resources quickly from one netlet to another. Clearly, this functionality is beyond the scope of a higher level virtual-network builder. It can be easily done using netlets.

Conversely, there may be application-specific knowledge that allows us to make certain operations *lighter*. For example, we may decide not to police the traffic in our netlet, while this is clearly unacceptable for virtual networks (as it might affect other virtual networks).

Note that even advance reservation of virtual networks for a time interval $[t_1, t_2]$ may not be part of a normal virtual-network builder, which means that either the resources have to be allocated immediately, i.e. *now* (and then not used for anything else until t_1), or we should wait and hope that when we try an immediate reservation at t_1, the resources will be available (i.e. no guarantees).

We have 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 [16]) between, for example, immediate reservations and advance reservations.

V. Call Admission Control

A basic CAC algorithm simply checks the reservation schedules to see if enough bandwidth is available to accept the new request and if so it updates the schedules. Depending on how requests are entered in the schedules, this may result in very poor resource utilisation. If, for example, the requests are entered based on peak rates, the result would be extremely conservative[8]. Note that statistical multiplexing for a time interval in the future is difficult due to the unknown behaviour of future flows.

Many CAC algorithms have been proposed in the literature. In section II we mentioned that a major problem of most of these existing methods is their reliance on a static model of traffic (in other words, accurate source characterisation). We propose CAC based on schedules and real-time traffic measurements that has no need for such a model. Bear in mind that the effective bandwidth (EBW) discussed here is the buffer service rate required to keep the cell loss ratio (CLR) due to queue overflow under a target bound[9].

A. Proposed CAC policy

The CAC algorithm is based on the traffic patterns observed during a time interval in the past. These patterns are used to estimate the traffic that will be generated if we include the new request. If this does not exceed the capacity, the call can be accepted. The effective bandwidth used for this is always less than (or equal to) the peak rate. Therefore, CAC using EBW gives better resource utilisation than CAC based on schedules alone[10]. At first sight, it seems that measuring traffic is unsuitable for our CA, because requests are made for an interval in the future: at admission time there is nothing to measure. On the other hand, we do have knowledge about the behaviour of current streams. This can be used for CAC in the near future.

The CAC in the CA is an attempt to bring together the strictness of schedules and the resource utilisation resulting from measurements. It may be thought of as a sliding scale with *measurements* at one extreme (corresponding to $t = now$) and schedules at the other (corresponding to $t = \infty$). Starting with CAC based on measurements and looking further into the future, we will see that the CA's CAC grows progressively more conservative (because we know nothing about the future streams, except their peaks). As time goes by we learn more about the flows in a specific interval that were admitted with the conservative CAC because some reserved calls will have started, so we can

[7] in this sense the relation between a netlet and a virtual network is similar to that between a thread and process

[8] but guarantees would be relatively hard

[9] It seems that the only QoS dealt with here is bandwidth corresponding to a desired CLR. However, QoS parameters are not orthogonal and other QoS parameters such as delay and delay variance also depend only on the probability distribution of the queue length (just like the CLR).

[10] the EBW estimators were developed as part of the *ESPRIT Measure* project, using some code written by Horst Meyerdierks

now make less conservative decisions for new requests for that interval.

B. Implementation

If we just consider schedules, the way the CAC might work when a new request comes in is as follows:

i. Let B_{total} be the total bandwidth on the resource.
ii. For the time interval that is specified for the new request, determine $B_{scheduled}$, the maximum amount of bandwidth reserved by adding all the peak rates of the reservations.
iii. Let b_{new} be the new request's peak rate.
iv. If $B_{scheduled} + b_{new} \leq B_{total}$ accept the request. If not, reject.

This is one of the most conservative schemes possible that works on a first-come-first-serve basis. The resource utilisation with this CAC will be low, but the resource guarantees are relatively hard. Next, we add measuring components to the resources. For switch ports, we add code that periodically sends the cell count per active connection to a *traffic server* which computes estimates for the effective bandwidth (EBW) of each of them (corresponding to a target CLR). It also obtains an estimate for the *aggregate* EBW.

In the schedules, we keep track of the EBW of the flows. Initially, the EBW is set to the peak rate. So, if the reservation is for time interval $[t_0, t_1]$ and *now* $\leq t_0$, we use the peak rate for CAC. For each new request, the CAC asks the relevant traffic servers for the EBWs of the active connections and also for the aggregate EBW.

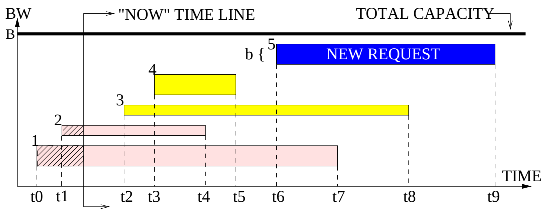

Fig. 8. A new reservation request arrives

The process is illustrated in figure 8. At $t = now$ a request arrives for bandwidth in $[t_6, t_9]$. The resource is a switch port. At $t = now$ there are 2 active connections (1, 2) of which we have EBW estimates. Connection 2 finishes *before* the new request starts, while the other one overlaps with the new reservation's interval. Between *now* and t_6 two more connections are set up (3 and 4) one of which finishes before t_6. Finally, we obtain the aggregate EBW. The CAC algorithm is now:

i. Let B_{eff} be the aggregate EBW that is used on the resource, $b_{eff}(n)$ the EBW of connection n and $peak(n)$ the peak rate of connection n.
ii. At $t = now$ a new reservation request r comes in for interval $[t_{start}, t_{end}]$ with $peak(r) = b$.
iii. Let E_{BW} be the estimated maximum of the bandwidth in $[t_{start}, t_{end}]$ (i.e. the bandwidth that we use to decide whether the request should be accepted or not). Initialise E_{BW} to B_{eff}.
iv. For all active flows x that finish *before* t_{start} do: $E_{BW} = E_{BW} - b_{eff}(x)$.
v. For all reservations y of which the connections have not started yet and which *overlap* with $[t_{start}, t_{end}]$, do: $E_{BW} = E_{BW} + peak(y)$.
vi. If $(E_{BW} + b \leq B_{total})$ accept, otherwise reject.

So for reservations in the future we use the specified peak rates, while for connections that exist already we can use the effective bandwidth estimated by the measure algorithm in the traffic server. Further in the future the CA will be more conservative in its admission control policy.

Traffic servers

Traffic servers are independent processes which talk to the switch on one end and the control architecture on the other. They receive raw statistics from the switch, process them and send updates of the EBW to the CA (on request). The traffic server is optional[11]. If a switch cannot provide the statistics, the CA still works (albeit more conservatively). In fact, it is conceivable to control a heterogeneous network where some switches (or indeed some of the ports of these switches) have traffic servers, while others do not. Also, the implementation may vary from switch to switch, allowing vendors to differentiate. The new set of components which includes the traffic servers is shown in figure 9.

Effective bandwidth estimation

Our CAC algorithm is based on [2] with few modifications. We briefly discuss some of the results. Assume our buffer is served at constant rate r. The workload process $W_t = \sum_{i=1}^{t} X_i - t \times r$ [12]. The queue length Q depends on W_t: $Q = max\{W_t : t \geq 0\}$. Under general conditions, a single-server queue has a queue

[11] and can be replaced at runtime
[12] $\{X_i\}$ is the number of arrivals in interval i (iid)

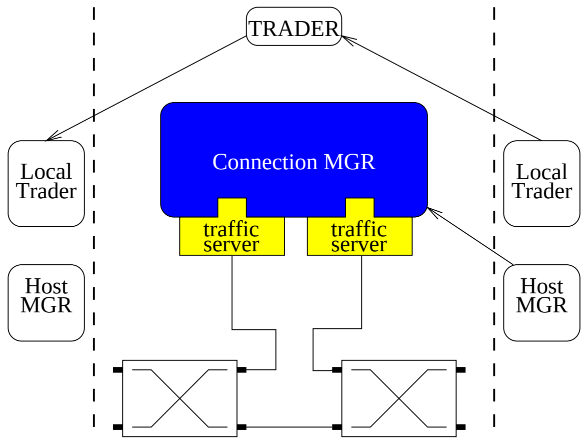

Fig. 9. Traffic servers have been plugged in

length distribution with asymptotes of the form:

$$P(W_t > x) \asymp e^{-tI(x)} \quad \Rightarrow P(Q > q) \asymp e^{-\delta q} \tag{1}$$

$I(x)$ is the rate function of the workload process and δ is called the decay-rate. Now, the r that corresponds to a CLR can be calculated as follows.

Define λ, a transform of I called the *scaled cumulant generating function* (SCGF), of the workload process as follows:

$$\lambda(s) = \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{sW_t}\right),$$

related to I by the Legendre transform:

$$I(x) = max_s \{xs - \lambda(s)\} \tag{2}$$

$r \times t$ is constant, so $W_t = \sum_{i=1}^{t} X_i - r \times t$, so:

$$\lambda(s) = \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{s(\sum X - r\times t)}\right) =$$
$$\lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{s(\sum X)}\right) - s \times r = \lambda_A(s) - s \times r$$

where λ_A is the SCGF of the arrivals process. So, given the arrivals SCGF we can calculate δ as function of r as follows:

$$\delta(r) = max\{s : \lambda_A(s) \le s \times r\} \tag{3}$$

We use this to find r, the service rate corresponding to a particular δ, i.e. to the target CLR. Using(1):

$$P(Q > q) \asymp e^{-\delta(r)q} \Rightarrow r = min(\delta(s) \ge -\frac{\ln CLR}{q})$$

and from (3) we know that $r \ge \lambda(\delta)/\delta$. Substitute $\delta = -\frac{\ln CLR}{q}$, so: $r = \lambda\left(-q^{-1}\ln CLR\right)\left(q^{-1}\ln CLR\right)^{-1}$ and since X_n iid:

$$\lambda(-x) = \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{-x(\sum(X_i) - xt)}\right) =$$
$$\lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{-x\sum(X_i)}\right) + xt =$$
$$-\lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{x\sum(X_i)}\right) + xt = -\lambda(x)$$
$$\to \text{so we find : } r = (\lambda((\ln p)/q))/((\ln p)/q) \tag{4}$$

The function $\lambda(\theta)/\theta$ is called the effective bandwidth. To estimate λ we use the following: for a large class of arrival processes it is possible to find a block length T such that the aggregated arrivals A_T are approximately iid. Then:

$$\lambda(s) \approx \frac{1}{T} \ln E\left(e^{sA_T}\right), \text{so } \hat{\lambda}(s) = \frac{1}{T} \ln \frac{1}{N} \sum_{i=1}^{N} e^{sA_T^{(i)}} \tag{5}$$

gives an estimation of the EBW. T should be large enough to make arrivals in that interval independent, but not so large that short bursts are smoothed out[13].

VI. State and Failures

Essentially, what happens when we make advance reservations of resources is that we introduce *state* in the network. There are a number of issues related to this state, one of which is the nature of it, i.e. how do we specify what portion of a shared resource we want to reserve (e.g. how much *bandwidth* we need for a connection)? There are two ways of doing this. We can take a simplistic approach, which says that we know nothing about our sources and the best we can do is provide the peak rates, or we can use a more sophisticated scheme where we supply a much more detailed characterisation of the source behaviour using parameters such as sustainable cell rate, burst length, cell delay variance, etc. For reasons mentioned in section (II), we have opted for the first approach, which gives us a very simple way of describing sources, but may lead to poorer resource utilisation. To counterbalance this, we used online measurements of traffic to estimate the effective bandwidth as discussed in section (V).

A. Soft state vs. hard state

Another issue related to the state in the network is the question whether it's *soft state* or *hard state*. RSVP uses the concept of soft state, which exists only as long as periodic messages are sent along the datapath [17]. If at some node, the messages fail to arrive,

[13] we have used $T = 200ms$

the soft state is removed. This is very attractive in certain aspects, but it may not be so attractive for advance reservations, because it requires nodes to be up all the time. This is probably an unreasonable requirement (when I reserve a video conference for next week, I don't care whether all nodes on the datapath go down overnight). Instead, we have opted for hard state, i.e. state that is only removed as the result of an explicit release operation. Note that hard state comes at the expense of more complicated releasing of resources (especially in the case of failures).

B. Failures and recovery

Failures can occur at any time and at any location in the network. The handling of failures and the recovery procedures in the CA are very simple but adequate for most applications. For example, the way the connection manager deals with the failure of an operation (e.g. a reservation or call setup failure, because a remote host went down) is by throwing an exception and releasing all resources allocated or reserved for this request. It also notifies all the host managers involved, so that they too know that the operation failed and can release their local resources or try again.

A more interesting problem arises when the connection manager itself crashes. We now have to think about what effect this should have on the connections. In our solution, we distinguish between two types of sessions in the CA: persistent and non-persistent. The non-persistent sessions crash with the connection manager, i.e. their connections will never be completed[14]. Persistent sessions on the other hand, are written to stable storage at reservation time. If a crash occurs before the start of the reservation interval and the connection manager comes up again before the start of the interval as well, then all reservation state is simply restored and the client is not even aware of the crash. Otherwise, if the connection manager comes up in the middle of a session, it figures out what connections should have been active at that moment, and if they are not active, sets them up. It also does all pending notifications and cleanup operations.

The most difficult failure, when a network is partitioned for a long time and the connection manager cannot reach one of the partitions to release resources, is the subject of future research. It is not difficult to handle if new connection managers come up in both partitions that have access to the same state database. This means that some sort of file replication is needed.

VII. Performance

We test the CA with a focus on the CAC algorithm. The ATM test-bed contains a number of Fore switches attached to HP, DEC Alpha and Solaris machines as well as some ATM cameras. Communication in the CA is based on IIOP or ANSA rpc.

Connection setup

Making a point-to-point reservation using unoptimised code takes about 7 ms on a Sun UltraSparc-1, not including the communication overhead. If we assume that the point-to-point reservation was made by a third party (i.e. the connection manager has to send SAP reservation requests to both the source and the sink), the total reservation time including communication is somewhere between 30 to 40 ms. This shows that almost all the time is lost in communication. The overhead incurred by setting up the call at wake-up time is null when compared to a CA without advance reservation. In fact, connection setup itself is now even faster, because initial communication and CAC overhead was already absorbed when the reservation was made. In a particularly slow implementation we measured an end-to-end connection setup time across a switch (where the connection is such that it can be joined to for a multicast) of several hundred milliseconds (which is clearly too slow for most purposes).

This slow implementation is useful, however, to examine the bottlenecks in the system. Basically, the reason why it is so slow is threefold. First, we use a slow implementation of IIOP[15]. Second, on the Fore switches (ASX-100, ASX-200) setting up multicast connections is particularly expensive, because the outport vpi/vci pairs have to be reserved for *all* other ports as well, so we suffer the reservation overhead multiple times. Third, probably the most important reason is the use of SNMP for communication with one of the switches (GSMP is used for the second switch). SNMP constitutes the bulk of the overhead. SNMP setup takes hundreds of milliseconds, while with GSMP 6-8 ms was achieved [15], so we conclude that SNMP is a bad choice for open signalling[16]. Moreover, we get doubly punished here be-

[14]If the crash happened before the start of the reservation interval, no connections were ever set up and never will be. Otherwise as soon as a connection manager comes up again, the connections' resources will be released.

[15]Each IIOP invocation takes 5 ms; these days latencies of 1-2 ms are achievable for commercial implementations

[16]the reason for using it at all is that all switches support it

cause, since our switches are not exclusively used for our research, we have to rely on placeholder connections in the switch to prevent other signalling (e.g. SPANS directly to the switch) to interfere with our virtual network. So for all connection setup operations on all ports, we first have to remove the placeholder connections (and set them up again on the teardown operation). In [15] a break-down of the overhead shows that a unicast setup time across a single switch of approximately 10 ms is achievable.

Admission Control

We submit a set of requests to the CA. The resource is a switch port shared by all calls. The relevant parameters are shown in figure 10. Note that the sum of the peak rates of source 1 (5000 cells/s) and source 2 (4000 cells/s) exceeds the capacity of the port (8000 cells/s). At $t = 0$, we try to reserve bandwidth for 3 calls, but since the CAC is based on peak rates only (no measurements yet), the request for source 2 (which overlaps with source 1) is rejected. Figure 11 shows the actual traffic of the flows[17].

TIME (s)	RESERVATION	INTERVAL	CAC RESULT
0	SOURCE 1	[1, 300]	ACCEPT
	SOURCE 2	[150, 300]	REJECT (!)
	SOURCE 3	[300, 450]	ACCEPT
50	SOURCE 2	[150, 300]	ACCEPT (!)

Resource capacity: 8000
Peak (source 1): 5000
Peak (source 2): 4000
Peak (source 3): 7000

Fig. 10. The requests, peak rates and capacity

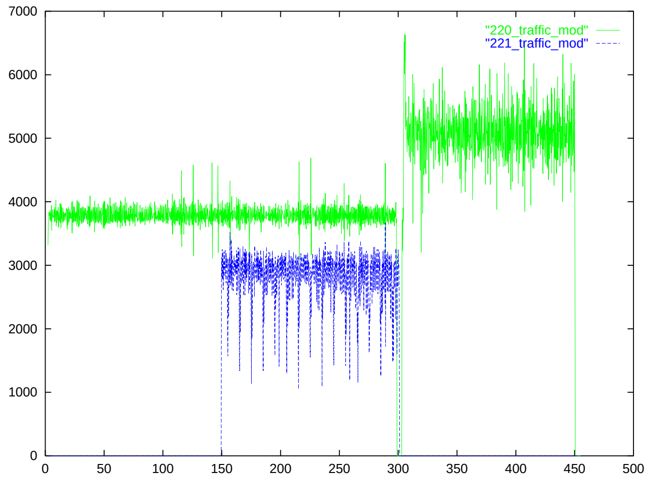

Fig. 11. Traffic on connections (bandwidth in cells/s against time in seconds)

The total traffic on the port is shown in figure 12. Also shown is the aggregate effective bandwidth which lies

[17]Note that although there are 3 connections, we only use 2 vci values. Since the third connection starts after the first connection has ended, it reuses its vci (220)

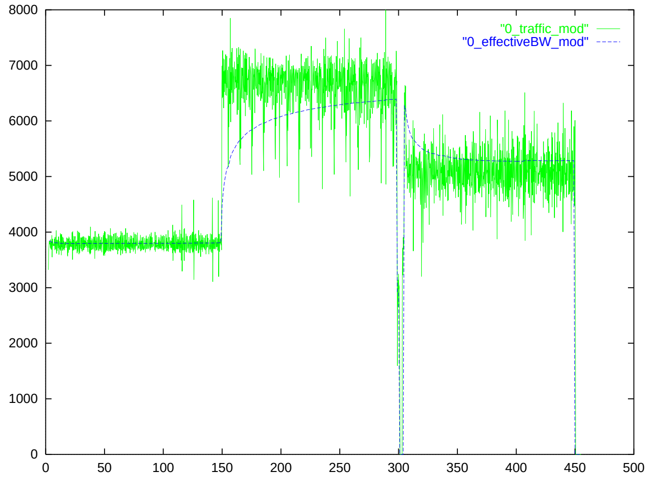

Fig. 12. Total traffic and effective bandwidth (bandwidth in cells/s against time in seconds)

well below the peak rates. Since our CAC uses the EBW of active calls rather than their peaks, it makes a less conservative admission decision after the first connection starts. So we see (figure 10) that when we try to reserve for source 2 again at $t = 50$, the request is now accepted. The EBW of the active connection (< 4000) plus the peak of the new request (4000) does not exceed the resource capacity. Figure 12 shows that the acceptance is justified—the total traffic never exceeds the capacity and resource utilisation improves considerably.

VIII. Conclusion

We have described a control architecture that allows for advance reservation of resources. This can be done in the form of complex connection patterns or in the form of netlets which basically are sub-virtual networks. Repartitioning virtual networks is easily done using virtual netlets. The admission policy merges traffic measurements with rigid scheduling. This results in good resource utilisation without the (almost impossible) requirement of accurate source characterisation.

References

[1] I. Cidon, T. Hsiao, A. Khamisy, A. Parekh, R. Rom, and M. Sidi, *The OpeNet Architecture*. Sun Microsystems Technical Report 95-37, December 1995

[2] S. Crosby, J.T. Lewis, I. Leslie, N. O'Connell, R. Russel, F. Toomey, *Bypassing Modelling: an Investigation of Entropy as a Traffic Descriptor in the Fairisle ATM Network*. Proceedings of 1st Workshop on ATM Traffic Management, 1995

[3] A. Dan, M. Kienzle, D. Sitaram, *A Dynamic Policy of Segment Replication for Load-Balancing in Video-on-Demand servers*. IEEE Multimedia, 1995

[4] M. Degermark, T. Kohler, S. Pink, O. Schelen, *Advance Reservations for Predictive Service.* Proceedings of NOSS-DAV'95, pp 3-14, April 1995

[5] R. Guerin, H. Ahmadi, M. Naghshineh, *Equivalent Capacity and its Application to Bandwidth Allocation in High-Speed Networks.* JSAC, Sept 1991

[6] S. Jamin, P. Danzig, S. Shenker, L. Zhang, *A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks.* Proceedings ACM SIGCOMM'95, 1995

[7] P.W. Jardetzky, C.J. Sreenan, R.M. Needham, sl Storage and synchronisation for distributed continuous media. Multimedia Systems vol. 3, September 1995

[8] R.J. Gibbens, F.P. Kelly, *Measurement-based connection admission control.* 15th International Teletraffic Congress Proceedings, June 1997

[9] A.A. Lazar, K.S. Lim, F. Marconcini, *Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture.* JSAC, Sept 1996

[10] G. Li, *DIMMA Nucleus Design.* APM Technical Report 1553.00.05, October 1995

[11] S.L. Lo, *A Modular and Extensible Network Storage Architecture.* University of Cambridge, Technical Report No. 326 (Ph.D. thesis), January 1994

[12] J. van der Merwe, I. Leslie *Switchlets and Dynamic Virtual ATM Networks.* Integrated Network Management V, Chapman & Hall, pp. 355-368, May 1997

[13] J. van der Merwe, I.Leslie *Service Specific Control Architecture for ATM.* Accepted JSAC, 1997

[14] S. Rooney, *The Hollowman: An Innovative ATM Control Architecture.* Proceedings of IM'97, May 1997.

[15] S. Rooney, *The Structure of Open ATM Control Architectures.* University of Cambridge, Computer Lab, PhD. thesis submission, February 1998

[16] A. Schill, S. Kuehn, F. Breiter, *Resource Reservation in Advance in Heterogeneous Networks with Partial ATM Infrastructures.* Proceedings of INFOCOM'97, April, 1997

[17] L. Zhang, S. Deering, D.Estrin, S. Shenker, D. Zappala, *RSVP: A New Resource ReSerVation Protocol.* IEEE Network, September 1993