– Controlling The Tempest – Adaptive management in advanced ATM control architectures

D.A. Halls and J.G. Rooney

Abstract

Recent research in ATM management has delegated control functions from the ATM switch into higher level *open control architectures* [1], [2], [3]. For example, functions such as Call Admission Control are implemented within such a control architecture rather than as an integral part of the switch. The association between the switch fabric and the switch control functions is made less strict, allowing, for example, a single entity to perform routing for a group of switches.

This greater flexibility in the provision of control is accompanied by an increase in complexity resulting from the possibility of making control decisions remotely from the controlled network element. The location of the controller in relation to the fabric is not just an implementation issue; it has a large impact on the way we handle traditional Operation and Maintenance (OAM) functions. For example, recovering from partial failure of the network is problematic when the controlling entity is communicating with the switch it controls across that same network. Most research into such control architectures has concentrated on the advantages of increasing flexibility; little work has looked at how the traditional techniques for creating robust networks can be adapted for use in this environment.

Mobile software agents can move to where their functions are carried out. When an agent has moved across the network it executes its function there. The function will not fail due to network problems. They can also reduce the amount of management state that needs to be carried across the network for control purposes. This is important when recovering from failure since the network is stressed and minimal expectations should be placed on its behaviour.

In this paper we show how mobile agents can be used within an open control architecture in order to implement robust Operation and Maintenance functions. We explain in some detail our work in implementing one such function.

Keywords

Adaptive control, Distributed control, Networks, Program interpreters

List of Figures

Both authors are with the University of Cambridge Computer Laboratory, Cambridge, U.K.

I. Introduction

SWITCH Independent (or Open) Control is characterised by a loosening of the association between the out-of-band switch control functions and the switch fabric. The control function's interaction with the fabric takes place across some well defined interface. This interface is as low-level as is consistent with its being independent of the switch's implementation. A server implements the interface by translating the switch independent functions into switch specific ones. The control functions can execute remotely from the fabric using a network transport to communicate with the server or alternatively they can execute "on switch" using some local transport.

We term the implementation of a group of out-of-band control functions a *control architecture*. At the Cambridge Computer Laboratory we have built several switch independent control architectures [3]. These use the Ariel switch control interface [2] for communicating with the fabric and depend on an Ariel server executing per switch. Our experimental network consists primarily of commercial ATM switches. We do not have access to the internal interfaces of all of these switches so we have implemented our Ariel servers using standard protocols supported by the switch, in practice GSMP or SNMP. [2], [4] have a full discussion of the issues involved. There are some overheads involved with implementing control functions in this way because the Ariel server is executing "off-switch", but the advantage of being able to control switches from different vendors with the same software is large enough that even within a commercial context such overheads may be worth accepting.

Therefore currently all our control functions are executed "off-switch". Once we have accepted the physical separation between the network and the control layer there is no need for each switch to have a controller at a distinct location, for example all the switch controllers can be executed at the same location. This greatly simplifies certain control functions such as routing. In [3] for example, a single connection manager is responsible for the creation, management and liberation of host-to-host connections for our whole experimental network.

Each time an application asks for a connection to be created it would not be reasonable for the control architecture to examine the state of the VPI/VCI tables and buffer allocation at all concerned ports, since this would involve the transfer of a great deal of state across the network for an operation which is required to have short latency. Instead, the control architecture learns about the network state at start-of-day and ensures its view of the network state is in phase with the network during its execution. This is not prohibitive, since every permitted state changing operation is by definition sanctioned by the control architecture.

The control architecture decides if a connection request should be granted by examining its local view of the network. Once the control architecture decides to admit the request, it instructs the switches to create the connection and updates its local view. During normal operation the switch should never refuse an operation that the control architecture has authorised, i.e. the control architecture has all the intelligence while the switch is relatively dumb.

Network failure is as possible within our advanced networks as in more conventional ones and for the same reasons. However, returning to a stable state after failure is more difficult due to the looser association between the control software and the switch fabrics. For example, the failure of a link could separate the controller from the network element it controls. Having a representation of the network's state in the control architecture requires having some means of resynchronising it with the real state in the event of such a failure.

The Laboratory's research has produced a framework — *the Tempest* — which allows many control architectures to run simultaneously over the same physical ATM network. This work is described in detail by our companion paper in this journal[4]. Each *virtual network* can be managed by different control software; the framework allows flexibility in the way control architectures are written, so the details of their operation cannot be predicted. The Tempest allows the control of virtual networks to be dynamically delegated to network operators. This allows greater flexibility in providing control but further exacerbates recovery when things go wrong; instead of one well-known resident control architecture being informed about failure, many foreign transient control architectures need to be notified. Moreover, handling of failure may vary considerably between control architectures.

Section II briefly discusses Cambridge's Tempest framework which supports delegated network control and virtual networks. It proposes using mobile agents to help synchronise control architectures with their switches' state. Section III describes a mobile agent that we have found useful in the Tempest for cleaning up network resources. Details of its implementation are given in Section IV. Related work is discussed in Section V and conclusions drawn in Section VI.

II. Network Control in The Tempest

A. Overview of the Tempest framework

The Tempest is a framework developed at the Cambridge Computer Laboratory for the execution of diverse control architectures simultaneously over the same physical ATM network. We outline the Tempest here, but it is more fully described elsewhere [2], [3], [4].

The Tempest consists of the following elements:

- an interface to switches called Ariel
- a partitioner of the resources of a switch between control architectures called Prospero
- a network builder which dynamically creates a virtual network for a control architecture by using Prospero
- a set of standard libraries to allow application developers to create control architectures

Figure 1 shows the relationship between these entities within the Tempest.

The control architectures executed within the Tempest may be generic (e.g. TINA-C, X-Bind, Q.2391) or application-specific, i.e. they can be developed with one application in mind and optimised for use with that application. The Tempest has the same role with regard to network resources that an operating system has to computer resources. Currently at the Laboratory we are running many different control architectures for a wide range of projects within the Tempest.

B. OAM in Open Control Architectures

Each of the Tempest control architectures has a representation of the state of the virtual network that has been allocated to it. For example, a control architecture may, for routing purposes, possess information about the current resource usage of the switches in the network. Each control architecture's view of the network state must be synchronised with the actual network state if effective control is to be achieved. As Section I pointed out, removing control functionality from switches increases the problem of maintaining synchronisation. We propose here using mobile agents to move control closer to switches while keeping the overall flexibility that open control architectures offer. This reduces the distance that switch state has to travel over the network and ensures that the agent processes state that is as up-to-date as possible.

In existing ATM systems, Operation and Maintenance (OAM) cells [5] are exchanged between neighbouring connected ATM switches in order to ensure the health of the physical link and the logical layers built over it. For example, if an OAM cell for a virtual path is not received after some time from the downstream switch, the OAM functions on the upstream switch may presume that the virtual path has been removed and take appropriate action. For instance, it could remove all the virtual channels associated with that virtual path in its channel tables and forward a failure OAM cell upstream. This scheme would allow all reachable parts of the network eventually to be informed about failure.

Within a network which supports Open Control we can still use the same mechanisms at the switch level. However, since the controlling software is not necessarily at the network elements through which the OAM cells are transported, the higher level control architecture also needs to be prompted to update its view. The switch can send some appropriate alarm type to the Ariel server, for example an SNMP trap, but how can the Ariel server notify the control architectures concerned?

One solution would be to have the Ariel server send alarm information to a central manager. This central manager could correlate the alarms and send appropriate messages to the control architectures concerned, e.g. remove all connections going through a port. This would be broadly similar to the current approach adopted within the ITU-T TMN standard as well as Internet network management.

Such a monolithic approach is conceptually simple but has practical drawbacks. [6] and [7] identify the weaknesses of this approach:

- poor scaling due to increase in communication and processing
- the need for a general-purpose API such as SNMP or CMIP in which many interactions are required to do simple things
- vulnerability to failure

Moreover, our control architectures are distributed entities that in general use the network itself for communication between their constituent parts. The manager may send an appropriate message to a part of the control architecture, but the control architecture as a whole still has to synchronise itself. If an alarm was emitted due to port failure, there is no guarantee either that the central manager can communicate with the control architecture or that the control architecture can get in contact with all of its parts. Failure to take this into account may make a bad situation worse [8]. For example, alarm notifications generated by the switches to indicate failure may overload the network, causing further failure and further notification. Operation and Maintenance functions need to be decentralised and adapt to the current network state if they are to be robust.

The OAM approach in the ATM standards is based on regular exchanges between control entities, with each controlling entity taking appropriate action locally when unexpected conditions are identified. Each part of the network will eventually return to a stable state independently, without specialised control messages being exchanged between arbitrary control entities. Open control architectures need to have a similar type of behaviour when handling failure. Since the control functions and the fabric are running on different processors, how can this be achieved?

We have been exploring the use of mobile software agents to solve this problem. The agents have a role similar to that of OAM cells in the ATM standards, i.e. they are transmitted between control entities at regular intervals using predefined and bounded network resources. The difference between the mobile agents and the OAM cells is that they can contain code as well as state. Resynchronising with the network does not require large numbers of high-level control architecture specific messages. All operations are carried out locally by the agent, ensuring that *if* the agent can be sent over the network then the operations will not fail due to network problems.

The use of mobile agents in this way is a generalisation of the current approach. OAM cells also carry programs, but programs written in a very simple language e.g. containing two sentences: *ok*, *not ok*. Our approach allows the programs carried between network entities to be written in a general purpose programming language. The ability to carry more complex programs is important because the Tempest does not specify how control architectures should be written. We cannot therefore predict the way that a control architecture wants its OAM cells to move from switch to switch. By using mobile agents for operation and maintenance, we allow control architectures to specify arbitrary policies for this.

We also allow them optionally to specify additional OAM functionality that is specific to their requirements. Control architectures written by trusted third parties can define and run their own agents above Ariel servers. This allows, for example, a control architecture to configure or reconfigure its VCI space by transmitting to the Ariel servers a single mobile agent which knows the required configuration; the number of control primitives executed across the network is consequently reduced.

The Tempest is suited to access of switch state by varied pieces of software because it provides a uniform, switch-independent interface to switches. Mobile agents can be written using only this interface to access switch state, eliminating the need to embed switch-specific code in them.

The next two sections of this paper present a mobile agent that we use in the Tempest framework for cleaning up network resources. Section III explains how it works and Section IV describes its implementation. This discussion demonstrates that a control architecture in experimental use can derive real benefit from using mobile agents.

III. Automatic Resource Freeing

A. Motivation

Within our network we are running many experimental control architectures. A control architecture which, due to programming error, only partially liberates or creates a connection pollutes the VCI space. Sometimes failure can occur in subtle ways. For example, programming error in handling concurrent connection creation requests can lead to erroneous connections being created. The control architecture's view of the network becomes out of phase with the real state. This is an example of the problem of desynchronization that was described in Section I, although in this case it is the control architecture rather than the network that fails.

The existence of erroneous connections may not become manifest until some time after their occurrence. Deciding whether the complete set of connections that a given control architecture possesses is valid or not requires correlating information from the virtual channel tables of all the switches that the control architecture uses. This task is laborious and better automated. It would have been possible to write a central entity which constantly polled the switches across the network in order to determine whether the VCI space for all control architectures was healthy or not. Our network is sufficiently small that this would have been feasible. However it is not a scalable solution as it involves the transfer of the complete virtual channel tables from all switches.

Consider a switch that has 16 ports, with a single virtual path used for connections and 256 virtual channel identifiers associated with the path. The number of distinct entries in the virtual channel tables is 4096. Assuming that each entry is a sextuple of integers, and all integers are encoded in 2 bytes, then the complete state of the switches' virtual channel tables takes up 48 kilobytes. On the other hand, the amount of code required to do the correlation is quite modest. We decided that it made more sense to move our program to the switch state rather than the other way around. Mobile agents offer a natural way of achieving this.

B. Mobile agents in the Tempest

When a control architecture starts up, the Tempest infrastructure can initiate one or more monitoring agents dedicated to it. These agents know:

- the resources allocated to the control architecture
- the location of the Ariel servers in the control architecture's virtual network
- the location of a control entity capable of determining if a connection is or isn't still in use

The mobile agents move between these locations using a dedicated virtual network, supplied by the Tempest infrastructure and shared between all control architectures.

The agent constantly moves between the Ariel servers and examines the state of parts of the virtual channel tables that have been allocated to its control architecture. The agent follows connections upstream determining if sinks on one switch have corresponding sources on the upstream switch. It only carries relevant information between the Ariel servers and does all correlation locally, thus reducing by several magnitudes the amount of information that needs to be carried around. When the agent recognises that a connection seems erroneous it moves to the location of the control architecture in order to have its opinion confirmed or denied.

Control architectures may have their own reasons for maintaining connections which appear invalid to the agent, so the agent cannot make the decision independently of the control architecture. For instance, the control architecture may leave redundant connections in place if pending new ones can make use of them. This can shorten connection set-up time. Another reason that the agent should liase with the control architecture is that it may have read switch state which was a result of an incomplete control architecture operation. The control architecture can recognise such cases and tell the agent that state it found to be potentially erroneous is in fact valid.

The simplest version of the agent flags an alarm to a human operator if the control architecture confirms that the connection should not exist. The human operator then takes appropriate action, for example to clean up the VCI space manually.

We then decided to extend our agent so that it automatically frees erroneous connections as soon as it identifies them. We call such agents Automatic Resource Freeing (ARF) agents. Such agents are useful during testing since they avoid the need for constantly stopping and restarting control architectures.

Figure 2 shows an example usage of an ARF agent. The agent is associated with a control architecture CTR-1, which has the VCI space 210. . . 220 allocated on the ports A, B and C of both switch 1 and switch 2. We assume all connections are made with a single virtual path identifier of zero.

The agent goes to the location of the interface to switch 1 and collects all the state associated with connections. It then goes to the location of the interface to switch 2 and correlates the information it has brought from switch 1 in order to determine if any of the connections are suspect. In the example, the connection on switch 1 from port A, VCI 210 to port C, VCI 219 (A,0,210→C,0,219) is suspicious due to the fact that A,0,219 is not in use on switch 2. The ARF agent then goes to the location of the interface to its control architecture and asks whether this connection is known. If not, it is removed.

The agent cannot delete the connection without first checking with the control architecture, since although the connection is clearly unusable it may be cached by CTR-1 for its own reasons. In general, before an agent modifies switch state, it must consult its control architecture because the latter knows about activity across the network as a whole. For example, although a switch mapping may not be in use for transferring data, the control architecture might want to keep it in place for re-use when setting up a new connection. Any activity that cleans up switch state must be coordinated with the appropriate control architectures.

If "ARFing" is done too quickly then resources are wasted; if it is done too slowly then the inconsistencies between switch and control architecture cause connections to be inexplicably refused. "ARFing" is made adaptive by varying the number of ARF agents executing and the frequency at which they move as a function of the number of inconsistencies found in the immediate past. When agents return to the control architecture, it can change their behaviour by modifying certain parameters, e.g. the amount of time they should wait at each Ariel interface. The control architecture can also launch more agents when appropriate.

The ability of ARF agents to move themselves means that the processing goes to the state, rather than the state to the processing (i.e. to the control architecture). Since ARF agents are small, this reduces the amount of network communication required to clean up resources.

In summary, our interest in the use of mobile code resulted from practical difficulties in managing a network running unreliable control architectures. This led us to consider the more general problem of the implementation of OAM functions, and in particular the problem of recovering from failure, within the Tempest. In Section II-B we proposed to use agents within high-level control architectures in a way analogous to the use of OAM cells within the standard control architecture. This section has motivated the use of mobile agents in this way. In the next section we give a detailed description of the implementation of the ARF function within the context of our research environment.

IV. Implementation

This section describes the system that we have built for running mobile agents in the Tempest. We first give a brief overview of the Tempest components involved and then of our mobile agent system. Finally, describe how they are interfaced with each other in order to test concurrent connection setup and to implement automatic resource freeing.

A. Overview of Hollowman

The Tempest can support many control architectures simultaneously. One such control architecture is the Hollowman [3]. The Hollowman is a general purpose experimental control architecture which devolves control from ATM switches into an application-level distributed processing environment. We use the Hollowman in the Cambridge Computer Laboratory to carry out control experiments.

The Automatic Resource Freeing (ARF) described in the previous section was initially run with a version of the Hollowman modified to create bad connections at arbitrary times. The speed and accuracy with which these erroneous connections could be identified was the benchmark against which we tested our implementation.

The Hollowman runs on DEC Alphas under Digital Unix, Sun SPARCs under Solaris and HP9000s under HP-UX. We used an SNMP version of the Ariel server running over ASX-200 Fore switches for our tests.

B. Overview of the Tube

The Tube [9] is a mobile agent system. It is able to send and receive marshalled expressions written in Scheme [10] to and from a network (see Figure 3). If tagged as executable, the expressions are passed to an interpreter for execution. The marshalling respects duplicate objects and cyclic references. We are able to ship the full range of expressions, including lexically-scoped functions and captured execution state.

This means that agents can create arbitrary functions and send them elsewhere, with their closing environments. Moreover, since we can transmit execution state, an agent can be stopped, moved and restarted elsewhere in a single call. The Tube does much more than just execute byte-code compiled scripts remotely — agents can modify themselves, create and dispatch other agents and treat their functions and execution state as first class, transmissible data.

Other facilities that the Tube provides are a restricted execution environment to prevent access by agents to sensitive resources, multi-threading using POSIX threads, a notice-board for agents to post messages for other agents to read, a user interface toolkit with state saving capabilities and the ability to display agents inside World-Wide Web pages.

The core state-saving functionality of the Tube is itself written entirely in Scheme. It is portable between Scheme interpreters and compilers, and thus also between operating systems. The current implementation of the Tube is Unix-based. It uses the Bigloo [11] Scheme compiler/interpreter, with patches to make it thread-safe. Versions of the Tube are running on DEC Alphas under Digital Unix, Sun SPARCs under Solaris, Intel Pentiums under Linux and HP9000s under HP-UX.

C. Interfacing the Tempest with the Tube

The Tube mobile agent system has been interfaced to the Hollowman control architecture. Because the Hollowman makes its functionality available at a high level, convenient encapsulation can be given to Tube programs.

The Tube has been linked with the Hollowman software library, using automatically generated stubs to glue the two together. These allow code running under the Tube to access the full functionality of the control architecture. Hollowman uses a CORBA [12] distributed processing environment for its internal communication; the Tube is isolated from this however — it just calls functions provided by the Hollowman stubs.

The Hollowman services used by the Tube are:

- importing offers (for data producers and consumers) from a trader
- invoking third-party connection over an ATM network between a producer and a consumer
- closing down ATM connections

These services are used in the Tube to send data over ATM connections between computers (see Figure 4). Each Tube execution site sets up a *producer* thread which takes care of sending data onto the ATM network and a *consumer* thread which reads data from the network. These threads are registered with the Hollowman's trader by using one of the functions in the Hollowman's software library. A program running on a Tube site which wishes to send some data to another site imports its site's producer thread offer from the trader and the destination site's consumer offer. It then invokes third-party connection setup between them. The specifics of how this is achieved and the trading facilities used are internal to the Hollowman.

At this point, an ATM connection is in place. The data is then given (via shared memory) to the producer. At the destination site, the consumer receives the data from the ATM connection and gives it (again via shared memory) to its Tube site for evaluation. The connection is then destroyed by the consumer. Any type of data made transmissible by the Tube can be sent in this manner, including executable code.

This procedure for sending Tube data over ATM networks using the Hollowman does not have to be handled explicitly by every agent – a function is provided which does it for them. Agents only have to specify that they want to send data to a Tube site via the Hollowman, give the name of the destination site's consumer offer and the function does the rest.

Making Hollowman functionality available in the Tube allows it to be called at run-time from an interpreter. This is useful for creating and manipulating connections by hand at a high level. The ability to send Tube data over an ATM network using the Hollowman allows the control architecture to be tested and resource freeing to be implemented using mobile code.

D. Testing the Hollowman

Mobile code has been used to test the stability of the Hollowman when many concurrent invocations are made on it from different locations in a network (see Figure 5).

In the test, a Hollowman-enabled Tube execution site is placed on each computer in an ATM network. An agent has been written which runs on Tube sites and can move itself between computers via the network by using the

Tube/Hollowman interface. It repeatedly moves around the network from host to host. At each stage, the choice of which host to move to is made randomly and the time before it moves on has a random element too.

By varying the number of these moving agents in the network, the Hollowman control architecture is tested at varying levels of concurrency. The more agents there are active in the network, the greater the number of connection requests handled by the control architecture at the same time.

This test was easily turned into a simple demonstration by giving each agent a graphical display which it carries around from host to host. Spectators type comments into the display, which are remembered by the agent as it moves around.

E. Implementing Automatic Resource Freeing

The low-level Ariel switch interface is used by agents to communicate with switches. Rather than defining static interfaces to higher-level functionality, using mobile code allows for a more flexible approach in which application-specific access is moved to the location of the switch's Ariel interface.

Automatic Resource Freeing (ARF) — using mobile code to check the consistency of switch state — has been implemented using the Tube and the Hollowman. For each switch in the network, a Tube execution site is created on the workstation closest to it (i.e. that has the shortest path to it or is directly connected to it, see Figure 6). These Tube sites make an Ariel function available to agents which returns the state of a switch via shared memory.

Mobile code to implement ARF has been written. Programs are sent out by the Hollowman control architecture and move around the ATM network, using the Tube/Hollowman integration to visit each switch's associated Tube site (see Figure 7). At each site, the switch's state is retrieved. Since the Tube provides state-saving facilities, each mobile ARF agent can carry about with it a view on the state of the network. When retrieving a switch's state, it can check for consistency against the state of the previous switch it visited. Inconsistencies must be checked with the control architecture before further action is taken because it has a wider view.

The following parameters may be varied by control architectures:

- the number of ARF agents active at any one time; each agent can leave behind when it moves a note telling other agents which state it has looked at and when it did so
- how frequently they move
- whether they clean up switch state themselves or simply notify a system administrator of inconsistencies
- how often they check with the control architecture for its view of network activity
- the frequency at which they execute; as more inconsistencies are found, this can be increased, allowing the process to be adaptive

ARF has been implemented using two mobile agents, one which cleans up inconsistent switch state and another that displays notification of broken connections (see Figure 8). In the figure, a multicast connection from the host *bailey* is valid for data transfer to the host *scilly* but broken in two other places (Port 3, VCI 214 and Port 8, VCI 216 on the switch *aynho*).

The sizes of the ARF agents are around 1 kilobyte. This overhead is offset because only one set of mappings has to be transferred between Ariel interfaces, whereas the states of all switches have to be sent to one location in a centralised solution. Processing is distributed and, given knowledge of network topology, the distances ARF agents travel can be minimised. ARF agents in regular use may be cached by Tube sites at Ariel interfaces so that they only have to be sent to them once. The cached agents may then exchange switch state amongst themselves.

Control architectures other than the Hollowman can implement their own methods for ensuring coherence between their views of network activity and the state of the switches. They can send to Ariel interfaces arbitrary resource freeing functionality that is consistent with how they control and manage their networks.

V. Related Work

A. Dynamically loaded control

These projects allow control functionality to be dynamically loaded onto switches, in order to provide locality of access to the switch and to allow the interface to switches to be evolved according to application needs.

The MAGNA project [13] has identified the deficiencies of a classical client-server based distributed processing environment as being inflexible and difficult to adapt and extend. It has proposed using mobile agents in order to add autonomy, mobility and adaptability to the TINA-C proposal [14] but has not described any implementation work. MAGNA discusses dynamically extending the control architecture with mobile code both to perform client-specific functionality and for management delegation. The Tempest uses connection closures [15] to do the former; this paper has presented an implementation of the latter — mobile code that is configured and initiated from the control architecture itself and which communicates with the Ariel switch interface directly.

Goldszmidt's Management by Delegation (MbD) project [7] uses a similar approach to ours. It uses *elastic processes*, which allow computation to be moved close to network resources by loading code onto devices. An elastic process is a process that can move (part of) itself. The term is language-independent, and describes an approach rather than an implementation. MbD is proposed for network management in order to provide locality of access, to overcome network latency and to add flexibility to the "rigid servers that provide data access capabilities to platform clients" [7].

Our approach is similar to Goldszmidt's in the sense that we use mobile code (elastic processes) to overcome network latency and make different resource access patterns available. However, we rule out moving new computation onto network elements themselves. Rather, our code sits on top of the Ariel switch interface, which provides uniform access to switch functionality and the ability to participate in multiple control architectures. Our code also sits below Tempest control architectures, which allows it to see and contribute to a more complete view of the network as a whole.

[16] proposes the use of mobile code in network management. Like this paper, it sees the benefits as being adding flexibility to the way resources are accessed and reducing network usage. However, it does not identify a specific problem or describe an implementation. We have used a mobile code system and an innovative ATM control architecture to implement a specific network control function that benefits from code mobility.

B. Active networks

Another approach is to make switches programmable by putting interpreters on them. Programs are sent along the data path and tagged for interpretation. Switches detect these programs and execute them as they pass by. This allows for intelligence to be down-loaded onto the switches themselves, achieving locality of access and the ability to change the functionality of services that a switch can offer. Netscript [17] uses this for "dynamic deployment of software at all nodes"; the Active Networks project [18] and Switchware [19] use this approach to allow data to carry their own networking algorithms with them.

Our mobile agents use a common, low-level control interface for communicating with switches. They achieve both locality of access, by moving code close to switches, and management flexibility in combination with the Tempest. We allow change in service functionality by running arbitrary control architectures above a partitioned network. We believe our approach extends naturally to loading code onto switches dynamically. However, at the time of writing the advantages of putting interpreters onto switches can be achieved whilst keeping network intelligence at a high level — allowing switches to be simple and inexpensive.

C. Adaptive management

These projects use numbers of mobile agents which move through the network monitoring and controlling its behaviour. The use of autonomous agents allows a decentralised approach to network management without using one single controlling entity.

The WAVE project [20] propagates "language strings" onto the network and interprets them at each node. Emphasis is placed on the lack of centralised supervision of these and the dynamics of such a system. One application that WAVE has been used for is intelligent management of open computer networks:

These structures may provide self-organisation and self-recovery from complex failures as well as forming the basis for integration of other (distributed and heterogeneous) systems. [20]

This is relevant to the work discussed in this paper because they have distributed interpreters throughout their networks. However, whereas our mobile code is part of, and launched from, an open ATM control architecture, they have "integrated with a usual management system by establishing two channels through Unix for transferring activation code and data between the systems in both directions" [21]. We argue that mobile code must have access to the internal interfaces of the control architecture and that use of mobile code in network management is best suited to non-critical maintenance procedures rather than for essential control and setup operations. Also, our work is based on ATM and mobile code has direct access to switch state; WAVE is implemented over TCP/IP and interfaces with a traditional network management system.

BT has investigated the use of mobile code in making networks robust and adaptable [6]. They have devised mobile agents that alter routing tables in order to manage load across a network and run them under simulation. We believe mobile code is suitable for carrying out such "background processing" in the network; indeed, our Automatic Resource Freeing mobile agent does this in a real network.

VI. Conclusions

The physical separation of control functions from the switch fabrics they control has an impact on the way we attempt to identify and recover from failure. The state of the network and the control layer's view of that state must be synchronised.

While it would be possible to use a centralised manager to coordinate resynchronising after failure, this has many drawbacks, particularly with regard to scalability and robustness.

The current means of implementing OAM functions within ATM networks is simple — involving the exchange of cells between neighbouring switches — and robust — each switch response to failure is local and makes no assumptions about the state of the network. We have explored the use of mobile software agents within open control architectures in a way which is analogous to the use of OAM cells within standard ATM.

We use agents to ensure that OAM functionality is close to switches and network usage is minimised. Since much control is delegated to control architectures, we allow arbitrary OAM code to run above a common switch interface rather than define static sets of OAM functionality there. Control architectures can ensure OAM functions take into account their views of network topology. They can also, if necessary, define OAM agents tailored to their needs.

We have motivated this discussion by explaining a function for Automatic Resource Freeing (ARF) which identifies erroneous network connections created by unreliable control architectures and removes them. This function is useful in our environment in which we are running many experimental control architectures. The motivation allowed us to demonstrate the structure and implementation of a network which used mobile agents for ensuring its health.

The efficiency with which Operation and Maintenance functions are executed is of much less importance than their need for robustness and making minimal demand on network resources. The overhead in executing interpreted code is therefore not critical. Of more concern is that the use of software mobile agents in the way suggested requires that every location in the network that they visit be capable of executing them. To some extent whether the approach defined here is useful will depend on the ubiquity of environments to run those agents and the security guarantees which those environments can enforce.

VII. Acknowledgements

We acknowledge the UK EPSRC for supporting this work through its studentships.

References

[1] Auriel Lazar and Koon-Seng Lim, "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture," *IEEE Journal on Selected Areas in Communication*, vol. 14, Sept. 1996.

[2] Kobus van der Merwe and Ian Leslie, "Switchlets and Dynamic Virtual ATM Networks," in *Proceedings of IM '97*, San Diego, USA, May 1997.

[3] Sean Rooney, "An Innovative ATM Control Architecture," in *Proceedings of IM '97*, San Diego, May 1997.

[4] Kobus van der Merwe and Ian Leslie, "Service Specific Control Architectures for ATM," *To appear in IEEE Journal on Selected Areas in Communication*, 1998.

[5] ITU-T, "ISDN-Maintenance Principles, B-ISDN Operation and Maintenance Principles and Functions," *ITU-T Recommendation I.610, ITU publication*, 1993.

[6] S. Appleby and S. Steward, "Mobile software agents for control in telecommunications networks," *BT Technology Journal*, vol. 12, no. 2, Apr. 1994.

[7] Germán Goldszmidt and Yechiam Yemini, "Distributed management by delegation," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, June 1995.

[8] Yechiam Yemini, "The OSI Network Management Model," *IEEE Communications, OSI Special Issue*, May 1993.

[9] David Halls, *Applying Mobile Code to Distributed Systems*, Ph.D. thesis, University of Cambridge Computer Laboratory, June 1997, Awaiting examination.

[10] William Clinger and Jonathan Rees (editors), "Revised(4) report on the algorithmic language scheme," *ACM LISP Pointers IV*, July–Sept. 1991.

[11] M. Serrano and P. Weis, "Bigloo: a portable and optimizing compiler for strict functional languages," in *Second Static Analysis Symposium*, Glasgow, Scotland, Sept. 1995, Lecture Notes in Computer Science, pp. 366–381.

[12] Jon Siegel, *CORBA — Fundamentals and Programming*, John Wiley and Sons, 1996, Object Management Group.

[13] T. Magedanz, K. Rothermel, and S. Krause, "Intelligent agents: An emerging technology for next generation telecommunications?," in *IEEE INFOCOM 1996*, San Francisco, USA, Mar. 24–28, 1996.

[14] W. J. Barr, T. Boyd, and Y. Inoue, "The TINA initiative," *IEEE Communications*, Mar. 1993.

[15] Sean Rooney, "Connection closures," *Computer Communication Review*, vol. 27, no. 2, Apr. 1997.

[16] Mario Baldi, Silvano Gai, and Gian Pietro Picco, "Exploiting code mobility in decentralized and flexible network management," in *Proceedings of the First International Conference on Mobile Agents (MA '97)*, Berlin, Bermany, Apr. 1997.

[17] Y. Yemini and S. da Silva, "Towards programmable networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, Oct. 1996.

[18] Dave Wetherall and David Tennenhouse, "The ACTIVE IP Option," in *Proceedings of the ACM SIGOPS European Workshop*, Connemara, Republic of Ireland, Sept. 1996.

[19] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie, "Switchware: Accelerating network evolution," Tech. Rep., CIS Department, University of Pennsylvania and Bell Communications Research, June 1996, White Paper.

[20] P. S. Sapaty and P. M. Borst, "An overview of the wave language and system for distributed processing in open networks," Tech. Rep., Department of Electronic and Electrical Engineering, University of Surrey, UK, June 1994.

[21] Hans-Thomas Götz, *The WAVE Project*, Dec. 4, 1996, WAVE project description World-Wide Web page.

Sean Rooney received the B.Sc. and M.Sc. degrees from Queen's University, Belfast in 1989 and 1990 respectively. After a period working at Alcatel Alsthom's research center at Marcoussis in France he is now working towards the Ph.D. degree at the University of Cambridge Computer Laboratory, U.K. His current research interests are network control and management.

David Halls received the B.Sc. degree from the University of Southampton in 1994, where he was sponsored by Matra Marconi Space Systems. He is currently working towards the Ph.D. degree at the University of Cambridge Computer Laboratory, U.K. His research interests include the potential applications of mobile code and document markup.

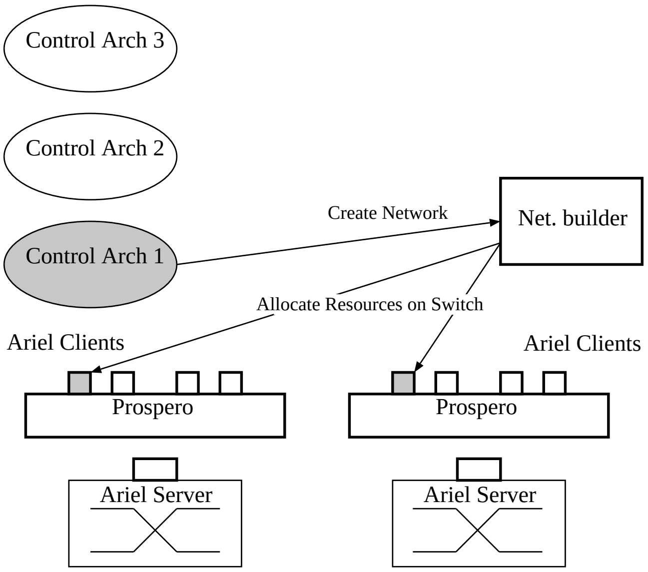

Fig. 1. The Tempest

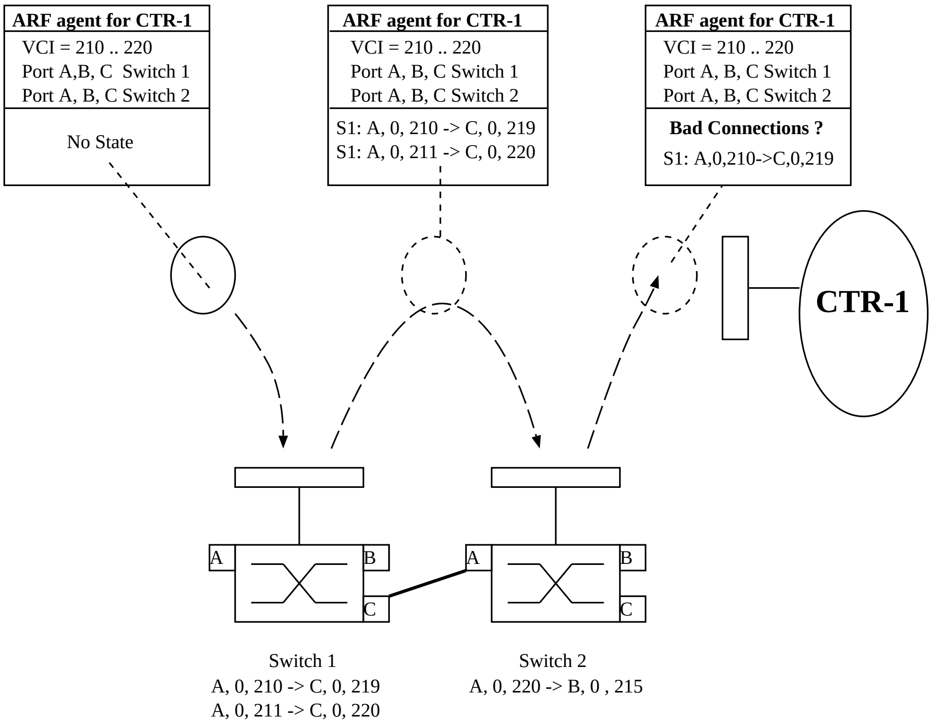

Fig. 2. Automatic Resource Freeing using mobile agents

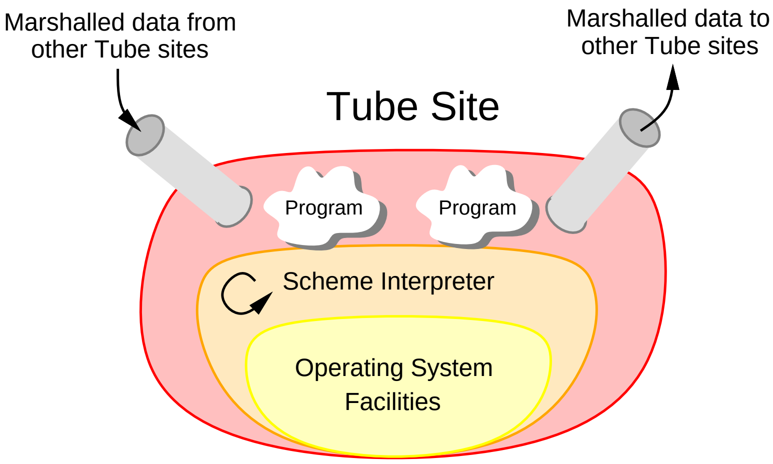

Fig. 3. Tube site

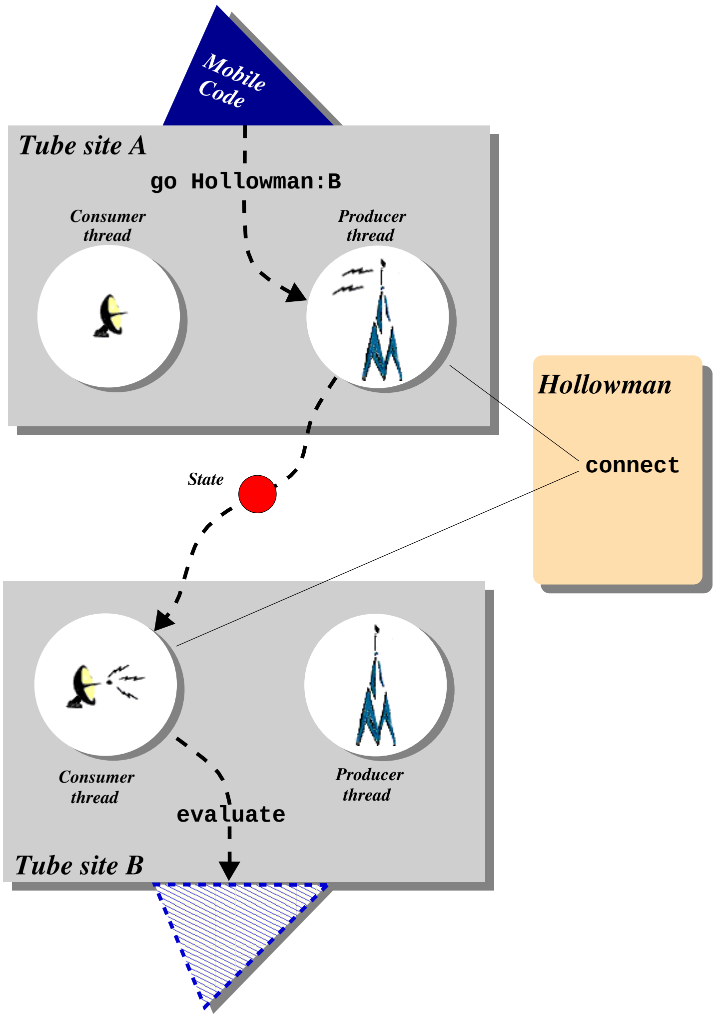

Fig. 4. Tube/Hollowman integration

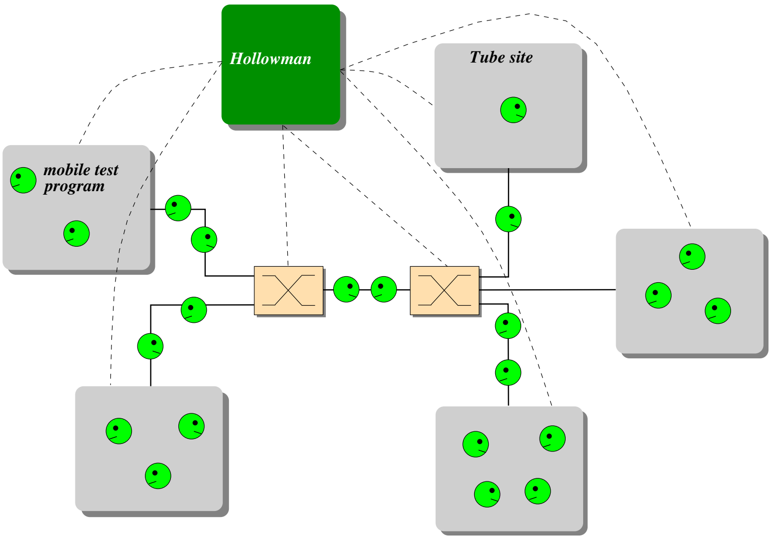

Fig. 5. Testing Hollowman using mobile code

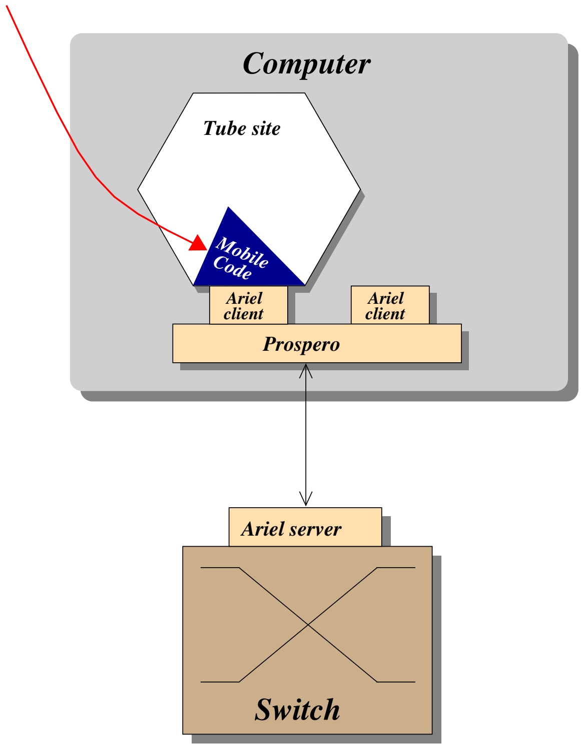

Fig. 6. The Tube and Ariel

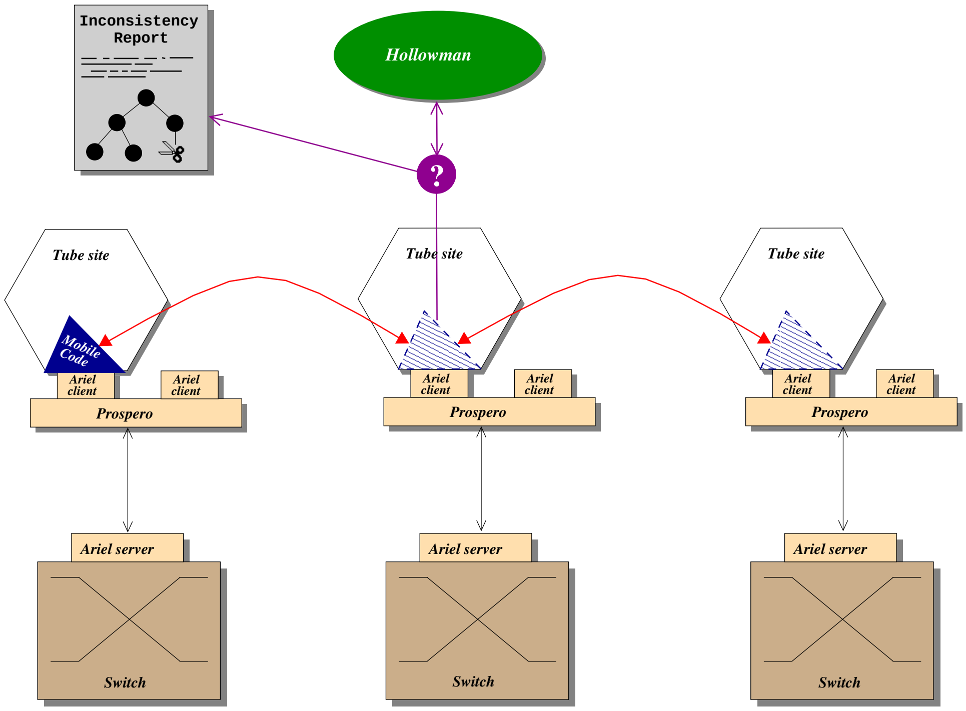

Fig. 7. Automatic Resource Freeing

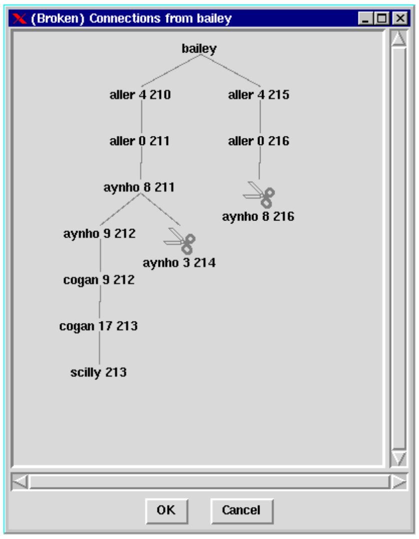

Fig. 8. Broken multicast connection