A novel architecture for active service management

I. W. Marshall, H. Gharib, J. Hardwicke and C. Roadknight
BTexaCT,
Adastral Park,
Martlesham Heath,
Ipswich IP5 3RE UK
ian.w.marshall@bt.com

Abstract

Future multiservice networks will be extremely large and complex. In this environment, Active Services will be needed to enable the rapid service evolution demanded by users. Active services also enable service management to be delegated to network users as a large set of independent small-scale management systems, thus minimising management costs. However, novel management solutions will be required to enable efficient multi-user management of the sites where the active services are run. We present an architecture for management of a network offering active services, which uses a combination of policies and adaptive algorithms to enable multi-user management of network based service components.

Keywords

Active Service, Policy, Adaptive Control, Autonomy, Configuration

1. Introduction

New Internet services and features are currently being introduced more slowly than users require, since existing human-intensive processes cannot cope with the rate of change. Active services [1, 2] are based on programs supplied by the users of the services. The programs run on devices owned by network operators or network service providers (such as caches, mirrors, conference controllers and firewalls). The aim is to enable users to have access to the services they require (custom services), whilst avoiding any requirement for operators and providers to manage large numbers of services. Active services should prevent the current problems being exacerbated by increased diversity of demand, but will not in themselves solve the current difficulties. A future network providing active services will be unbounded in both scale and function, since an enormous range of services will develop and evolve at an unprecedented rate. In order to fully realise the intended flexibility it will be necessary to combine active services with a highly automated management and control system. A ‘global state’ of the system will be impossible to ascertain due to its massive scale. In cases, such as this, conventional methods of control and

management do not apply and adaptive methods of control must be used [3]. In this paper we provide a brief description of our active service network implementation, based on an extended version of the application layer active networking (ALAN) proposal [1]. This is followed by a description of the management architecture we have designed for ALAN. The first part of the architecture description provides details of a flexible, policy driven system for the creation and distribution of management information using a combination of policies, messaging and active programming. The second part describes how the degree of automation of the system can be increased using an adaptive control mechanism. This uses a novel distributed algorithm, partly inspired by observations of bacterial communities [4]. Finally we present some initial results that demonstrate the effectiveness of our proposal. The management system we describe is to our knowledge the first to successfully combine, policies, active programming and adaptive control, as we believe is required for future active service networks.

2. ALAN

ALAN [1] is based on users supplying java based active code (proxylets) that runs on edge systems (dynamic proxy servers - DPS) provided by network operators. It is assumed that many proxylets will be multiuser, and most requests will be to "run" a proxylet that already exists in the network. Messaging in the extended version uses HTML and XML and is normally carried over HTTP. ALAN is primarily an active service architecture, and the discussion in this paper refers to the management of active programming of intermediate servers. Figure 1 shows a schematic of a possible ALAN node. The arrows represent the possible information flows. Objects should be thought of as peers (no explicit hierarchy is intended).

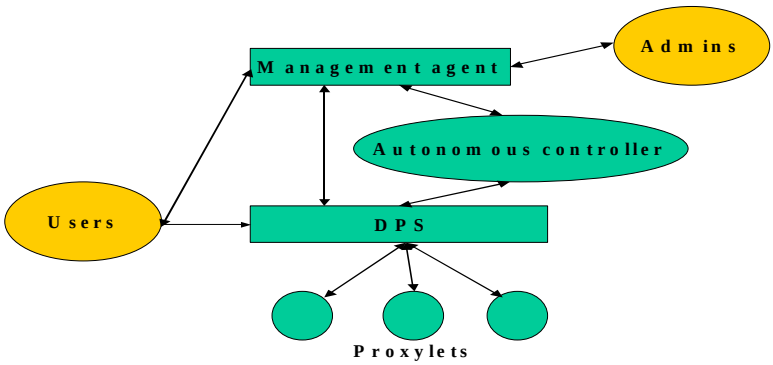

Figure 1. Schematic of proposed ALAN node design.

We have designed and partially implemented an active management solution for active service networks based on role-driven policies [5,6]. The management system

supports a conventional management agent interface that can respond to high level instructions from system operators. This interface is also open to use by users (who can use it to run programs/active services by adding a policy pointing to the location of their program and providing an invocation trigger). Typically the management policies for the user programs are included in an XML metafile associated with the code using an XML container, but users can also separately add management policies associated with their programs using HTTP post commands. In addition the agent can accept policies from other agents and export policies to other agents. Our system provides an extensible monitoring and configuration service that enables users to specify their configuration, monitoring and notification requirements to network devices using policies. Each policy specifies a subject (the policy interpreter), a target list (the objects to be changed if the policy is activated), an action list (the things to be done to each target), a grade of service statement, and the authorisation code, id and reply address of the originator. The policies are named using a universal policy name, which is also part of the policy. The names currently take the form *upn:originator id.subject.target_list.last_modified_time* and are likely to be globally unique. The actions take the form of conditionals, some of which will be true on first reading and false thereafter. Others will be true when some trigger event occurs that the policy interpreter can detect. The policies can carry enclosures (e.g. the code required to execute an action, or a pointer to it) embedded in the action list, so we describe the management system as 'Active'. The enclosures can obviously be instances of active services, i.e. proxylets. We refer to policies containing active services as service execution policies. Normally an execution policy would contain a pointer to a proxylet rather than the proxylet itself, but the policy and the proxylet will often be part of the same notification, especially if the notification is a request to run a service originating from an end user. A notification is implemented as an XML entity that must contain a policy (or a pointer to an appropriate policy if the policy was sent before) with at least one action such as ‘store the enclosed data’ or ‘run the enclosed service’, and an appropriate data enclosure. The enclosure may include several additional policies targeting different local entities, the code (or pointer) for a proxylet, or an event report. This extends the usual TMN definition of notification to include all management information that needs to be disseminated, rather than just event reports. The notifications are multicast to relevant hosts (using an appropriate anycast or multicast address), where they are received by a management agent, and any enclosed policies are stored in a local policy store if the appropriate key is present (i.e. a key associated with a role authorised to supply policies to the target device). The management agent has an extensible table of authorisation policies to enable this decision. Roles are allocated using a public key infrastructure. If a notification addressed to the management agent encloses a number of component policies, each component policy must specify the subject (normally an object oriented program) intended to use it as part of their rule-base. The local policy store has a table of policies for each registered subject and the management agent will store the component policies in the appropriate parts of the database.

Our approach avoids many information handling problems by using a lightweight scalable mechanism for notification transfer. The Information Management System [7] consists of a hierarchy of '*store and forward*' notification stores, with notifications being classified by their propagation characteristics and storage duration. Two types of these information stores are used, their selection depending on the complexity of querying required against storage availability. While simple but fast stores offer a load balancing and traffic controlling function, more complex stores permit management information analysis.

The DPS also has an autonomous control system that performs management functions delegated to it via policies (scripts and pointers embedded in XML containers). This autonomous control system is intended to be adaptive, and is integrated with the conventional agent by sharing policy stores.

Not shown in the figure are some low level controls required to enforce sharing of resources between users, and minimise unwanted interactions between users. There is a set of kernel level routines [8] that enforce hard scheduling of the system resources used by a DPS and the associated virtual machine that supports user supplied code. In addition the DPS requires programs to offer payment tokens before they can run. In principle the tokens should be authenticated by a trusted third party. At present these low level management activities are carried out using a conventional hierarchical approach. We hope to address adaptive control of the operating system kernel supporting the DPS in future work.

3. Management agent services

The management agent receives notifications from any entities that send them, and interprets the policy named in the notification wrapper. If authorised by the sending entities' role (identified in the wrapper policy) it stores any embedded events in the event log, and any embedded code in the proxylet cache. In addition it places any embedded policies in the policy store, informs the subject(s) and associated management facilities of the update to the store and generates an event recording the change in an event log. The subject will then activate the policy and either enact it immediately or await the trigger condition. For example the policy may specify the management agent must record each usage instance for a particular interface that it supports. The default target is the local event log so the agent must forward the usage data to an event logger, and provide the event logger with a policy for handling the data (this policy would be embedded in the action list of the first policy). The action to install the embedded policy is immediate, but the action to forward data is triggered by use of the interface. When data is forwarded the event logger will generate an event from the base data, adding a timestamp, a sequence number, an event class, a time to live, the source object name (management agent) and the generating policy name(s). The agent must also inform any entity nominated in the triggered policy that the event has occurred, using a notification service. The address list is identified using a name server. The name server maintains a list of subjects,

targets, policies, and manager ids registered at the local node. Each list resolves to the address(es) of the entities. In the case of managers, such as the agent, the address could be a multicast address the manager obtained from the information management system.

Figure 2 illustrates the overall design of the Event and Notification services supported by the management agent at a single node.

A *Notification Provider* is any entity (including other management agents) that requests the export of information in the form of notifications. The receiving agent (a notification consumer) distributes any policies embedded in notifications to the local *Event Service Elements* or *Notification Service Elements* as required. There can be any number of local Event/Notification Service Elements in the system, some of which could be proxylets (or groups of proxylets) that implement the custom management services required by VPN customers. Based on the policies supplied, the Event Service Elements generate events and pass them to the Notification Service Elements, which in turn dispatch them to *Notification Consumers* identified by the notification policies. A *Notification Consumer* is any entity that accepts notifications.

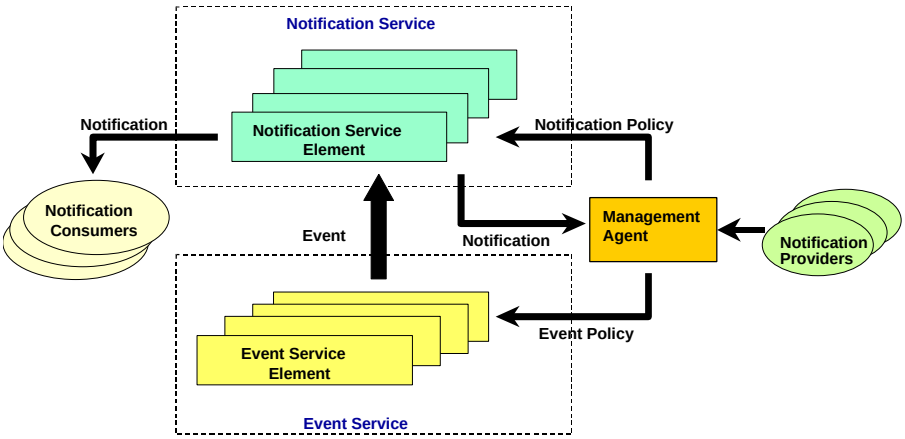

Figure 2. Relationship between Event and Notification Service Elements.

It is possible for an Event/Notification Service Element to receive multiple policies whose targets overlap. In this case, for each target the service element needs to determine which policy takes precedence over the others. This is achieved by considering first the local scope of role of the policy creator and then the local scope of the policy targets. A policy whose creator has higher authority (at the object where the conflict occurs) will take precedence; for instance, a policy created by a manager will usually supersede one created by a user, if the manager has configuration authority over the object, but the precedence order can be altered for each object if required (using a further policy generated by the root administrator for the object). For policies whose creators have equal role scope, the one whose scope of targets is

larger (i.e. includes the targets specified by the other) will take precedence. In circumstances where the scope of targets overlap, but one is not a superset of the other, the overlapped set will be the target of both policies.

3.1 Event Service

The Event Service is realised via Event Service Elements (ESEs) as shown in figure 3. Operation of the service is initiated when an authorized entity, such as a user or administrator, requests the monitoring and generation of events by sending an event policy to one or more ESEs. The policy can contain information such as:

- The entities and attributes that need to be monitored
- The processing (e.g. aggregation, averaging, threshold detection, etc.) required
- Event destination (logged locally or exported in the form of notifications).

In addition, it is possible to perform actions on a policy already running in an ESE:

- De-activate: stops all interpretation of a policy.
- Activate: activates a de-activated policy.
- Delete: deletes a de-activated/active policy.

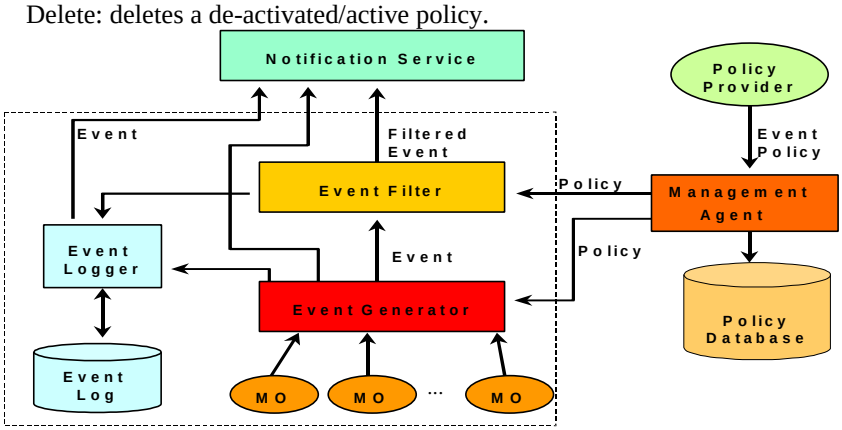

Figure 3. Components of an Event Service Element (within dotted line).

The role of each component is now described.

- MO: Managed objects whose attributes are being monitored to detect occurrences.
- Event Generator: polls managed objects periodically to detect the occurrences. Alternatively, the objects may send occurrence reports asynchronously to the event generator. According to the event policy, once the Event Generator has formatted the occurrence into an event, it either stores the events in the *Event Log*, sends them to *Event Filter* or delivers them directly to the Notification Service.
- Event Filter: receives events and filters them according to the event policy.

- Policy Database: The active, deactivated and deleted policies are held in the policy database. The change of status (e.g. from active to deactivated, etc.) for each policy is time stamped.
- Event Logger: Stores the events sent to it in the event log (a database)

Each event policy contains sub-policies for each component of the Event Service Element. It is possible that some of these sub-policies might be null. An event policy has the following form:

if *subject* then
    load Sub-Policy-1 to Event-Generator
    load Sub-Policy-2 to Event-Filter
endif

Here *subject* identifies the management agent (acting for an ESE) that is expected to receive and store the policy. Therefore, the management agent should first check whether it is the intended subject before commencing loading the sub-policies. It is also possible that the same policy may need to be sent to more than one ESE. In this case, the policy sender may use a *group* name for the subject. The group name will represent all the intended policy recipients. The policy can be distributed to the recipients via multicast. Upon receiving a policy, the policy receiver should check whether it is a member of the group represented by the group name.

Each event will carry the following information:

- Time-stamp: indicates event creation time.
- Event Sequence Number: uniquely identifies each event generated within an ESE. The sequence numbers generated by different ESEs may overlap.
- Management Agent Name: identifies the agent that has generated the event.
- Source Distinguished Name: identifies the attribute/resource about which the event is reported.
- Attribute Type: indicates the type of the attribute about which the event is reported. For instance, integer, character, etc.
- Attribute Value: value of the attribute, interpreted according to the attribute type.
- Time-to-live: indicates the length of time (after event generation) that the event is valid. This is used to mark those events that lose their informational value after a period of time. If an event is received after its time-to-live has expired it can be deleted without any processing.
- Policy class/id: identifies the policy resulting in the generation of the event.
- Requesting Manager: names the entity, which has issued the policy resulting in the creation of the event. This is useful, for example, when the event recipient itself has not requested the creation of the event.
- Version Number: identifies which version of ESE generated the event.

A key type of event records payments as its attribute value, and associates payments with the policy that resulted in the payment. Such events are stored in a special section of the event log known as the payment log.

3.2 Notification Service

The Notification Service is responsible for transmission of management information and associated data between management agents. The information could be event reports, policy distributions and updates, service code distributions and updates, or any combination of these. The Notification Service is realised via Notification Service Elements distributed across the system. Figure 4 illustrates the components of a Notification Service Element.

The role of each component in the diagram is now described.

- Information Receiver: receives management information from an information source (e.g. an ESE or a policy generator), or extracts it from a log if there is an appropriate authorization policy. The information is forwarded to the Notification Filter component.
- Notification Filter: filters information received by the information receiver according to the criteria specified in a notification policy.
- Notification Encryptor: encrypts the information according to the notification policy. The policy can either refer to the default encryption algorithm provided by Encryptor, or alternatively, provide its own algorithm.
- Notification Wrapper: adds a notification wrapper.
- Notification Dispatcher: receives notifications from the Notification Wrapper and sends them to destinations identified by the notification policy. The notifications may also be logged locally.

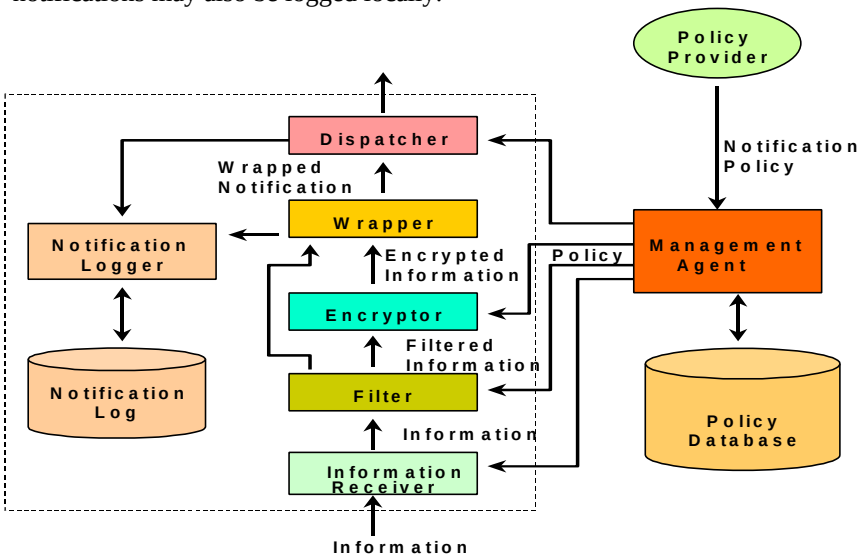

Figure 4. Components of a Notification Service Element (within dotted line).

Each notification policy identifies the Notification Service Element (NSE) that it is destined for, and also contains sub-policies for each component of the NSE. Some of these sub-policies might be null. A notification policy has the following form:

if *subject* then

load Sub-Policy-1 to Event Receiver
load Sub-Policy-2 to Filter
load Sub-Policy-3 to Encryptor
load Sub-Policy-4 to Dispatcher

endif

Here *subject* identifies the management agent (acting as an NSE) that is expected to receive and interpret the policy. The information provided by the sub-policies for the NSE's components are as follows:

Information Receiver: Identification of which entities are allowed to send information to the Information Receiver. The policy may specify that the information reciever should extract the information from a named datastore such as a log.

Notification Filter: Any processing/selection which needs to be performed on information. For example, it may be required to report only a single event of a certain type during a time-period to some consumers.

Notification Encryptor: The encryption algorithm to be applied.

Notification Wrapper: Whether a recipient should acknowledge the receipt of notification, and the priority level to be assigned to each notification type.

Notification Dispatcher : List of notification recipients, and their notification attributes, e.g. immediate/deferred, transport protocol choice, time to live

Each notification can be considered to consist of two parts, a data part and a notification wrapper part. The notification wrapper is an XML container with the following mandatory tags:

- Management Agent Name: uniquely identifies the agent (and NSE) that has generated the notification.
- Notification Sequence Number: uniquely identifies each notification generated by a NSE.
- Reply Needed Flag: indicates whether the recipient is expected to return an acknowledgement.
- Priority (low, medium, high): indicates the notification's priority. This can be used to categorise the notifications and determine the resources required for their processing. For instance, high priority notifications may be placed on high priority output queues, and hence, transmitted quicker.
- Version number: identifies which version of the Notification Service has generated the notification.
- Policy location: The policy may be part of the data enclosure or in the policy store of the receiving agent.
- Policy name: Points to the policy for processing the notification. Any data enclosures in the notification must conform with the implicit expectations of the policy's action list. Normally the data enclosure is also an XML document

4. Autonomous controller

The system described above will rapidly generate large numbers of policies and as it grows, the need for user intervention will tend to grow even faster. It is thus imperative to combine the policy based management approach with a significant improvement in management automation. Given the nature of the problem domain this can only be done using adaptive control. Conventional control of dynamic systems is based on monitoring state, deciding on the management actions required to optimise future state, and enforcing the management actions. Adaptive control [3] is based instead on learning and adaptation. The idea is to compensate for lack of knowledge by performing experiments on the system and storing the results (learning). Commonly the experimental strategy is some form of iterative search, since this is known to be an efficient exploration algorithm. Adaptation is then based on selecting some actions that the system has learnt are useful using some selection strategy (such as a Bayesian estimator) and implementing the selected actions. Unlike in conventional control, it is often not necessary to assume the actions are reliably performed by all the target entities. This style of control has been proposed for a range of Internet applications including routing [9], security [10,11], and fault ticketing [12]. As far as we are aware the work presented here is the first application of distributed adaptive control to service configuration and management.

Holland [13] has shown that Genetic Algorithms (GAs) offer a robust approach to evolving effective adaptive control solutions. More recent work [14] has demonstrated the effectiveness of distributed GAs using an unbounded gene pool and based on local action (as would be required in a multi-owner internetwork). In addition Ackley and Littman [15], demonstrated that to obtain optimal solutions in an environment where significant changes are likely within a generation or two, the slow learning in GAs based on mutation and inheritance needs to be supplemented by an additional rapid learning mechanism. Our bacterial algorithm [4] is a distributed GA with an additional rapid learning mechanism, and forms the basis of the adaptation performed by the autonomous controller in our architecture. In this paper we aim to identify the role of autonomous control in our policy driven management system and describe how the autonomous controller is integrated and provide only a brief sketch of the bacterial algorithm.

One of the most distinctive features of bacterial genetics is the process of plasmid interchange, in which one bacterium accepts copies of genes exported by another. This process is in effect a learning mechanism, and enables bacteria to acquire new capabilities (such as antibiotic resistance) extremely rapidly. In our controller we treat policies as though they were genes, and policy exchange between entities as plasmid interchange. If the controller is programmed (like a bacterium) to autonomously export policies that improve its performance, and de-activate policies that degrade performance, useful policies will spread and poor policies will cease to be executed (until conditions change).

In fact the controller monitors all the execution policies in the policy database that name it as subject, and autonomously deactivates those that are generating the least revenue (as recorded in the payment log). The payment events are recorded in the log by the event service. In order for a service to be autonomously controlled its execution policy must include an action that loads a polling policy into the event generator in addition to the expected actions 'do procedure' or 'run service instance'. We assume that control will not be needed for all actions, only those with a high resource cost, so we only apply control to execution policies, i.e. policies that contain at least one action with the semantic 'load and run a programme'. We also assume that services (or some other entity specified in the polling policy that can provide payment on behalf of the service) will be polled for payment, as this allows the charging regime to be localised. In addition the controller exports to its immediate neighbours in the network graph (via the notification service) the policies generating the most revenue when the node fitness function (revenue - cost) is high. Exported policies have one or more remote autonomous controllers as subject. A receiving management agent simply stores them as de-activated policies in the appropriate part of the policy store. Whenever the autonomous controller deactivates a policy it will examine all the deactivated policies and activate a random selection, to compensate. It will also inform the originator of the policy that it has been deactivated (if the user defined the appropriate grade of service when he wrote the policy). This allows user to increase the level of payment and avoid permanent deactivation if the policy has a high priority. The autonomous controller has two further capabilities: it will shut down the DPS if fitness has been low for some time, and copy the DPS to a nearby vacant site if fitness has been high for some time. Policies that are never useful will tend to disappear completely, since nodes that possess them will be more likely to shut down. Policies that are useful for some demand but not for others will persist but may not always be activated. Since the active services can only run if a policy pointing to them is triggered or interpreted, this effectively means that useful active services will spread, and useless ones will disappear, i.e. service deployment, configuration and withdrawal have been automated.

The autonomous controller is thus acting as a configuration manager, distributing policies/services (remember services or pointers will be embedded in notifications containing execution policies) to where they are needed and activating them on demand, without needing any knowledge of what the demand is or what the policies represent. It is also acting as a low-level account manager since all the policies it controls, that point to services, will not execute unless payment events are generated by the service they point to. This is very convenient since an active services network must respond rapidly to the introduction of new services, enabling them to spread to wherever there is demand, whilst providing a stable quality of service for existing services. When a user develops a new proxylet, or an improved version of an existing proxylet, he should not be required to identify all the locations where it should be stored and/or run. Typically the user lacks both the time and the knowledge to make such a decision for himself and in any case cannot predict demand from other users of his program. At the same time if a user introduces a new

service he should not be able to access his service until he has paid the appropriate fee. In our system the user introduces the policy to a default home server (e.g. the site of the web cache he is using) and the placement/distribution of the service is then fully automated. Given that the number of DPSs will be large, and the number of proxylets unbounded, the correct configuration algorithm will be one, like ours, that needs as little human/manual intervention as possible, as the manual optimisation of proxylet placement soon becomes untenable.

5. Experiments

To test our approach, and demonstrate the automated deployment, distribution and withdrawal of a service using only the autonomous controller, an event service and a notification service, we simulated a community of 400 DPSs, each of which was controlled by an adaptive management agent as described in the architecture above. The network supported a range of 8 active services and the traffic distribution was random in both space and time. An execution policy enabling a new service (service h) was then introduced to one node (using a notification), along with some simulated, network wide demand for the service provided by the corresponding proxylet. Initially the users did not offer payment for this new service. It can be seen (Figure 5) that for a brief period some requests are handled (hence the success rate is briefly >0) but the network soon fails to execute any of the requests for the new service. This is because any proxylets for this service are not earning their DPS any revenue (no payment events are generated), and the execution policy is therefore being replaced by more lucrative policies. This illustrates how the autonomous controls deal with the introduction of malicious code intended to defraud the operator. Once the users start to pay for the new service the request dropping rate decreases and the proxylet autonomously spreads around the network using the notification service.

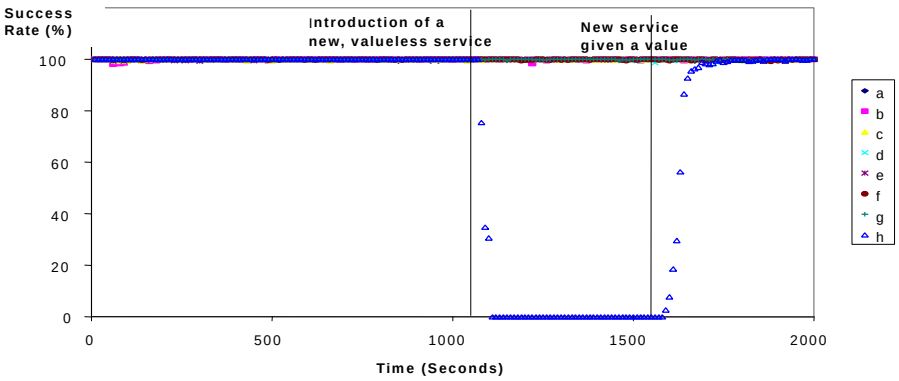

Figure 5. Service provisioning consequences of introducing a new service.

Figure 6 shows the results of the same experiment expressed in terms of handling latency. When the service gene for service h is introduced, a few requests are handled without affecting the latency of existing services, and the new service quickly dies

out as no users are offering payment. Subsequently no latency is given for the new service as no nodes are processing it (all the requests are dropped). Later, when users start paying for the service there is a period of high latency as the plasmid distributes around the network but soon the latency of the new service is similar to that for the other services.

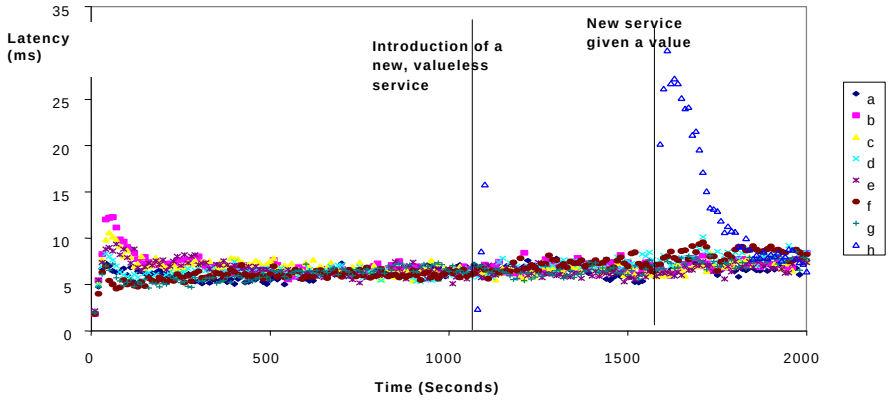

Figure 6. Latency Consequences of introducing a new service.

In figure 7 we show the average request drop rate across the network of bacteria and compare the performance with a number of alternative methods of distributing the active services. The alternatives are:
a) Random static placement of services at network nodes
b) Caching of requested services with a random replacement algorithm (Cache I)
c) Caching using a least recently used replacement algorithm (Cache II)

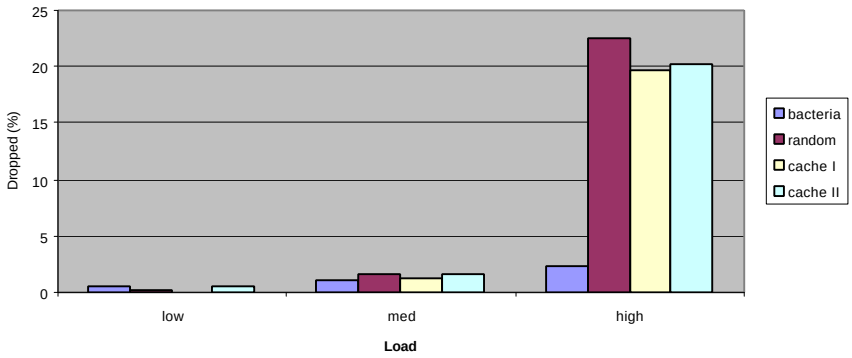

Figure 7. Request drop rates for different distribution mechanisms.

The tests were performed at loads of 10% (Low), 40% (Medium) and 80% (High). At low loads all the algorithms offer similar performance levels. As might be expected, at medium and high load our algorithm is a significant improvement over

random placement. More surprisingly it also significantly outperforms caching. We believe this is due to the small size of the caches. Each cache holds up to eight services (4 live and 4 paged out - the same as the bacteria). This is intended to represent the number of proxylets that can be held in the RAM of a low spec PC, such as might be used in a commodity based cluster at a network server farm.

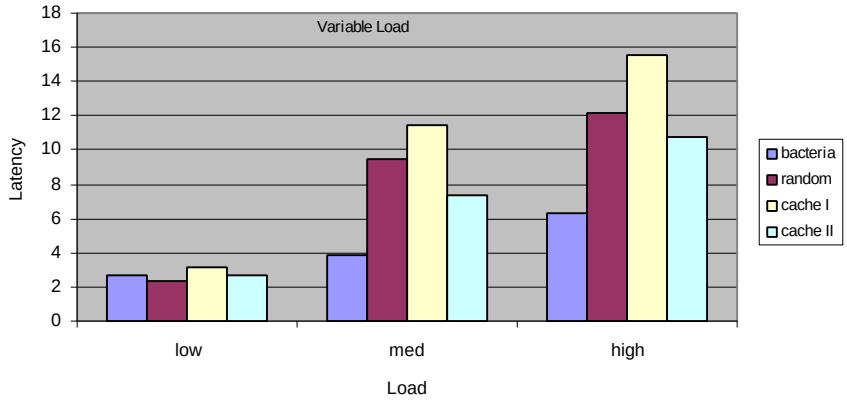

Figure 8. Average latency of several approaches to distributing active services.

Figure 8 shows the average end to end latency experienced by service requests in our modelled network (expressed in milliseconds), and compares it with the latency experienced using the alternative active service distribution mechanisms listed above. As before the adaptive bacterial approach is as good as the other alternatives at low loads, and is clearly an improvement over the best alternative (standard LRU based caching - CacheII) at medium and high loads. The simple experiments we have shown illustrate that our system can automate key aspects of performance, configuration, account and security management of the services in an active service network. It is also clear that the network will adaptively work around faults until they either die, or are manually repaired.

6. Discussion

There has been an extensive study of policy based management undertaken within the IETF [www.ietf.org]. Unfortunately the policy schema used in this work is not sufficiently expressive to meet the needs of an environment with multiple managers, such as an active network, since it does not allow policies to be linked to the role of the originator. We have instead based our notion of policy on the work of Sloman et.al. [5]. Our policy schema differs from theirs in a number of ways, in particular;

a) we do not distinguish between authorization and obligation since many of our policies can do both depending on the context in which they are applied
b) we express constraints as part of the action statements and replace the

constraint field with a grade of service field. This is to allow managers to customize their priorities for policy handling, and ensure key policies are acted even when the system is overloaded

c) we express policies in XML – a purely pragmatic decision.

The notification and event services were initially implemented in CORBA [16], but we found performance was poor and coding was hard. Since the ethos of active services is to make creation of new services easy we moved to XML on the basis that coding is straightforward and lightweight. We have found [17] performance in wide area contexts is also rather better since the messaging is asynchronous and there is therefore a low probability of blocking whilst awaiting delayed replies. Of course CORBA now provides good support for asynchronous messaging too (it did not at the time of the original work) but there is no benefit to be gained from a further rewrite in CORBA.

The autonomous controller has not yet been implemented, since we are still tuning the algorithm and attempting to identify how much we can usefully manage using it. We plan a full implementation in the near future.

To the best of our knowledge, our integration of the bacterial control algorithm [4] with conventional management services is entirely novel. This integration enables the retention of well developed and understood techniques for those aspects of active service management that require control by the operator, whilst enabling highly automated control to be implemented on behalf of end users (who would not wish to control the system in detail). Crucially the application demonstrated enables low cost deployment and withdrawal of a large range of new services, as envisaged in an active network. We do not feel this is possible using conventional hierarchical manual control. In addition we feel the autonomous control of policies is an extremely promising solution to the explosion in the number of managed entities that policy based management entails

7. Conclusions

Active networks will require extensive use of adaptive control techniques, particularly where user supplied code has to be managed. The most obvious control point is the DPS in the ALAN approach. An autonomous adaptive control architecture for dynamic proxy servers in an active network has been proposed, based on combining a novel genetic algorithm inspired by observation of real bacterial communities, with policy based management techniques and active programming of management systems. Simulations have shown that the proposed system can handle key aspects of fault, configuration, account, performance and security management successfully in a large scale, dynamic environment.

8. References

[1] M. Fry and A. Ghosh “Application Layer Active Networking” Computer Networks, 31, 7, pp. 655-667, 1999.

[2] E. Amir, S. McCanne, R. Katz, “An active service framework and its application to real time multimedia transcoding” Computer Communications review 28, 4, pp178-189, Oct 1998.

[3] Y.Z. Tsypkin. "Adaptation and learning in automatic systems", Mathematics in Science and Engineering Vol 73, Academic press, 1971.

[4] C.M.Roadknight and I.W.Marshall, "Adaptive management of an active services network", BTTJ 18, 3, Oct 2000

[5] Sloman M., "Policy Driven Management for Distributed Systems", Plenum press Journal of Network and Systems Management, Plenum Press

[6] Lupu E., Sloman M., " A Policy-based Role Object Model", Proceedings of the 1st IEEE Enterprise Distributed Object Computing Workshop (EDOC '97).

[7] Bates J., Bacon J., Moody K., and. Spiteri M., “Using Events for the Scalable Federation of Heterogeneous Components”, *Proceedings of 8th ACM SIGOPS European Workshop*, Sintra, Portugal. September 1998.

[8] D.G. Waddington and D. Hutchison, "Resource Partitioning in General Purpose Operating Systems, Experimental Results in Windows NT", Operating Systems Review, 33, 4, 52-74, Oct 1999.

[9] G. DiCaro and M. Dorigo, "AntNet: Distributed stigmergic control for communications networks", J. Artificial Intelligence Research, 9, pp. 317-365, 1998.

[10] D.A. Fisher and H.F. Lipson, "Emergent algorithms - a new method of enhancing survivability in unbounded systems", Proc 32nd Hawaii international conference on system sciences, IEEE, 1999

[11] M. Gregory, B. White, E.A. Fisch and U.W. Pooch, "Cooperating security managers: A peer based intrusion detection system", IEEE Network, 14, 4, pp.68-78, 1996.

[12] L. Lewis, "A case based reasoning approach to the management of faults in telecommunications networks", Proc. IEEE conf. on Computer Communications (Vol 3), pp. 1422-29, San Francisco, 1993.

[13] J.H. Holland, "Adaptation in Natural and Artificial Systems" MIT press, 1992.

[14] R. Burkhart, "The Swarm Multi-Agent Simulation System", OOPSLA '94 Workshop on "The Object Engine", 7 September 1994.

[15] D.H. Ackley and M.L. Littman, "Interactions between learning and evolution". pp. 487-507 in Artificial Life II (ed C.G. Langton, C. Taylor, J.D.Farmer and S. Rasmussen), Adison Wesley, 1993.

[16] J.R.Fallows and I.W.Marshall "A CORBA assisted multimedia proxy server" in "Multimedia applications services and techniques" ed Hutchison and Schafer, LNCS 1425, pp149-162, Springer Verlag 1998

[17] I.W. Marshall et. al., "Active management of multiservice networks", Proc. IEEE NOMS2000 pp981-3