A Policy based Management Architecture for large scale active communication systems

Ian W Marshall and Paul Mckee

BT Adastral Park, Martlesham Heath, Ipswich, IP5 3RE
ian.w.marshall@bt.com, paul.mckee@bt.com

Abstract. An initial design for a policy based management system combining conventional hierarchical control and significant local autonomy is described. A critical part of the design is a scheme of partial guarantees for policy distribution and execution. This scheme renders explicit the non-determinism that is implicit in policy based control schemes that include conflict resolution, and to some extent replaces the need for conflict resolution. Some preliminary implementations of the design are described, and implications for further work are discussed.

Keywords: Policy, Autonomy, Guarantees, Store, management requirements

1. Introduction

One of the key aims of next generation communication networks is to support the dynamic business relationships demanded in the new economy, for example by activities such as e-commerce. Aside from introducing obvious features such as mobility support it is crucial that operators create management and support systems that allow rapid response. Since human operators are the principal bottleneck in existing systems this means that network operations and business processes must become more distributed and more automated. Recently active services [1-4] have been proposed as a means of making the network more responsive to users demands. However this approach has so far not addressed management automation, and in addition creates new management challenges. The new challenges include that of supporting multiple, user based, management domains making independent decisions and using a wide range of technologies. Centralised control will not be possible and the response of the system will be the result of a collection of autonomous actions. There is however the need to exchange management information and policies between interacting systems. In order to achieve this we require common information models or at least a common information syntax. We are developing [5-11] an approach based on events (for monitoring the state of system components) and policies for expressing the desired behaviour of system components. The approach embeds a novel autonomous controller [12,13] that enables increased automation of some key low level management functions. To support heterogeneous systems it is essential that the information and policies are represented in a platform neutral way. We propose the use of XML, a de facto standard for the representation and exchange of

text based information, particularly where the information needs to be automatically processed. XML allows users to define representations specific to their own applications with a well defined formal syntax. In our approach we propose the definition of management data and concepts using XML DTD's or Schema. The use of DTD's or schemae gives a reference representation of information that has a consistent meaning in multiple systems. Ideally these representations would be standardised, but where this is not possible automated transformation may be used to transform policy and event formats to domain specific representations. XML based policies and events may be propagated between nodes in the system using commonly available Internet protocols ensuring the greatest possible reach of the proposed technology, but no single protocol is mandatory. Our current experiments have used XML to distribute policies that control the mobility of code in an active network application. Previous papers [5,6] have discussed the use of XML as a resource description approach and we are currently working on resource management monitoring using XML. One advantage of the use of XML and either DTD's or schema is the automatic checking for well formedness and validity that occurs when an XML document is parsed, allowing early rejection of erroneous input. The use of XML as an intermediate representation still allows policies to be developed using any existing management tools - they will just be transformed prior to transmission.

In this paper we present for the first time the details of the policy handling architecture required by our XML based management system. The architecture relies on a novel categorisation of policies, that enables us to support both dependable and best effort management activity. This is crucial since we cannot provide guarantees that all requests for management action emanating from users can be satisfied by the available resources. The paper also describes a preliminary implementation of key aspects of the architecture. In the next section we summarise ALAN, the infrastructure we are attempting to manage. This is followed (sections 3 and 4) by a high level description of the management architecture, and how policies are handled within it. In section 5 we motivate and describe the policy categorisation scheme, and give examples of its application. Section 6 presents our initial implementation and results.

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 uses HTML/XML and is normally carried over HTTP. There are likely to be many DPSs at a physical network node. It is not the intention that the DPS is able to act as an active router. 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.

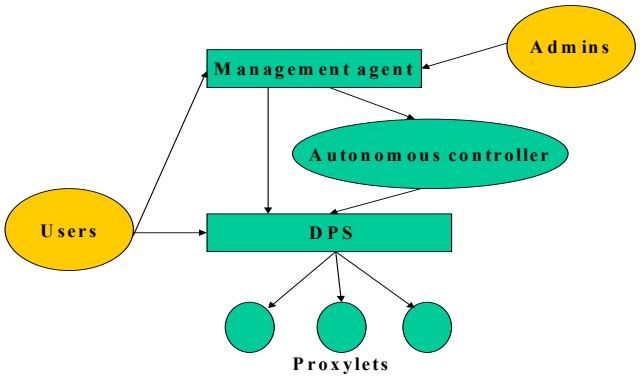

Figure 1. Schematic of proposed ALAN node design

We have designed and partially implemented an active management solution for multi-service networks based on role-driven policies [7,8] and Application Layer Active Networking (ALAN) [9]. The management system supports a conventional management agent interface that can respond to high level instructions from system operators [10,11]. 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. We refer to the metafiles as macropolicies since they may embed many statements concerning a range of entities. Individual IF-THEN-ELSE statements are referred to as atomic policies. For our purposes a policy must consist of one or more atomic policies referring to a single entity.

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) and the authorisation code, id and reply address of the originator. The policies can carry enclosures (e.g. the code required to execute an action, or a pointer to it), so we describe the management system as 'Active'. The enclosures can obviously be instances of active services. An event is simply a policy with a single action - "store the enclosed data", and an appropriate data enclosure. 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 policies are multicast to relevant hosts, where they are received by a management agent, and 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. A policy addressed to the management agent can also enclose a number of component policies, each of which specifies the subject (normally an object oriented program) intended to use it as part of their rule-base. The policy store has a table of policies for each registered subject and the local 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 policy transfer. The Information Management System [10] consists of a hierarchy of '*store and forward*' policy stores, with policies 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 [12] 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 o/s kernel supporting the DPS in future work.

3. Management Architecture

Previous publications have described our overall architecture for the management of multi-service networks [5] and a proposed architecture for active server management [11]. This paper describes the design and partial implementation of an event service element designed to use XML as both an event description format and as a policy and object metadata format [6]. This event service element is designed to run on a single physical platform, or to form part of a larger distributed system on multiple platforms. This system provides an extensible monitoring and configuration service that enables users and managers of the system to specify their requirements to system components using policies. The basic building blocks of our current implementation are shown in the following diagram.

The system is designed to be as flexible as possible and its operation is initiated when policies are provided. There are a number of different providers of policies that may interact with the system, every software component or user program installed on the system will have an associated metadata description that will be supplied to the event service element. This metadata description will contain a number of policies that pertain to the operation of that component or user program, it might be perhaps considered as a meta-policy. In a similar way the hardware upon

which the system is running will have a metadata file containing policies that control for example access to hardware resources. End users and remote mangers may both supply single policies via the communications adapters and other event service elements may pass policies to any other element. At this stage we are assuming nothing about the structure of any policy except that it has a header in the XML description that describes it as a policy.

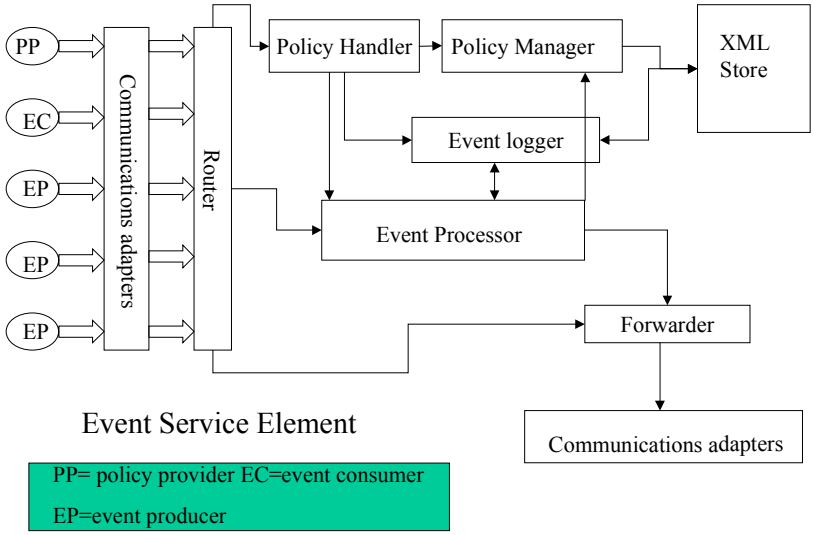

Fig.2 Schematic representation of an event service element

Once identified as a policy the incoming information is routed to the policy handler where the policy will be parsed and checked against the document type definition (DTD) that describes its structure. Again this imparts flexibility for the introduction of a range of new policy definitions to a live system. The policy handler also determines whether the policy is destined to manage the operation of the event service element or should be stored in order to be able to handle detected events. Management policies within the event service element are used as a means of traffic management, events may be aggregated and filtered in the event processor, and the time scales for any locally stored events may be modified.

In this initial prototype implementation the structure of the events is again left as flexible as possible. Events are received at the communications adapter and within the router the header will identify the XML as an event. It will then be passed to the event processor where it will be parsed and checked against a DTD, if no DTD is supplied the event is just checked for well formedness. This again accommodates future expansion in the types of events that may be specified and collected.

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 as described earlier. We have recently developed a novel bacterial GA described elsewhere [13] that 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 policies that it controls in the policy database, and autonomously deactivates those that are generating the least revenue (as recorded in the payment log). In order for a policy to be autonomously controlled it must therefore include an action such as 'Get_Payment'. The controller also exports the policies generating the most revenue when the node fitness function (revenue/cost) is high. Exported policies are addressed to the autonomous controller so the policy receiver simply stores them as de-activated policies. Whenever the autonomous controller deactivates a policy it will examine all the deactivated policies and activate a random selection, to compensate. 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.

The autonomous controller is thus acting as a configuration manager, distributing policies/services 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 are generating payments (and will not execute without payment events being generated). 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. Given that the number of DPSs will be large, and the number of proxylets unbounded, the correct configuration algorithm will be one that needs as little human/manual intervention as possible, as the manual optimisation of proxylet placement soon becomes untenable.

5. Policy Classification

Clearly, to support the hybrid management system we have designed we must be able to classify policies such that those that are appropriate for autonomous or adaptive handling are clearly identified. It is obvious that in order to enable the autonomous controller to take control of those policies sent to it, as identified above, the policies cannot be atomic. Policies addressed to the autonomous controller must encapsulate atomic policies for the entities it is controlling. However, it is also necessary for administrators to have a thorough grasp of which atomic policies can be allocated to autonomous control in this way. One reason is that the level of guarantee offered by the autonomous controller is weaker than that offered by a completely deterministic system. In addition if we are to retain the spirit of XML it must be possible to write policies behind tags that can be ignored by receiving entities that do not support the tag for local policy reasons. With this in mind we have attempted to devise a scheme allowing policy authors to specify the grade of service their policy requires. These specifications could be expressed as constraints in the existing policy notation of Sloman et. al. [7]. We would prefer to express them as a top level categorisation since this makes it more efficient to store the policies in a structure based on level of guarantee. Such a storage approach simplifies the task of conflict identification considerably by reducing the number of entries that must be considered. The basic classification is illustrated in table 1, and deserves some explanation.

The three rows represent different levels of guarantee concerning whether the action specified in a policy will be carried out. The highest class represents the case where the policy will always execute and corresponds to the grade of service assumed in most existing policy architectures. Clearly business critical actions specified by operators should normally be in this class. The second category assumes that policies are sent to many network elements and offers a guarantee that the policy will execute at some of the entities it is sent to. This is the level of guarantee offered by the autonomous controller, and is probably appropriate to many information gathering actions, particularly when multiple execution returns redundant information. In addition this class is ideal for the enhancement processes associated with intermediaries in an active network, since they typically are required once and once only on the path taken by the traffic being enhanced. The key benefit is that managers can specify actions without any need to decide where they should execute. In other words load balancing and conflict avoidance are automated. The third class emulates the level of service offered by the Internet and is intended mainly for end users (whose policies will also often get overridden by a conflict resolver in a more conventional system). It may also be of benefit to operators who can specify an experimental action without needing to worry about the impact on live traffic, since the action will only execute when load is light.

	Guaranteed correct (Account and Security)	**Errors tracked** (Configuration)	**Best effort correctness** (Fault and Performance)
Guaranteed execution wherever specified	Critical actions (e.g. authorisation changes)	Most configuration actions	Critical event logging and notification
Guaranteed to execute somewhere and/or sometime	Sample based billing	Some Proxylets	Most notifications and policy distributions
Best effort execution	Intrusion tracing proxylets	Active service composition, proxylets	Most event logging

Table 1. Examples of management activities to which the different grades of policy handling might apply

The three columns represent grades of service for a more complex concept. Correctness can refer to execution, distribution or persistence. It is envisaged that the notion of correctness will be context dependent, i.e. it will depend on the target, subject and action specified in the policy, together with any constraints. For the case where the action requires storage of information, such as an event, the three classes can be interpreted as follows. The first column represents the event being treated as a transaction and stored using a distributed transaction controller. This grade should only be used for requirements (e.g. charging?) where accuracy and consistency are paramount. Policies in the second column will trigger an exception message to the policy owner if the storage or execution fails, but will make no attempt to rollback system state. This grade is ideal for many configuration management records, since existing processes already tolerate inconsistency arising from unavoidable human errors at installation time. Policies in the third column will simply fail, or store incorrect data. This is ideal for most non critical management information collection, particularly where the information is only used for off-line analysis or data mining. In cases where storage of results are not required, correctness could apply to persistence of the policy being correctly maintained for the time specified in the constraints, or alternatively to distribution of follow on actions to the correct targets.

A policy in the second row of the second column will thus guarantee to execute once and tell at least one owner whether or not it succeeded. On the other hand a policy in the second row of the first column will distribute messages wherever specified whenever it executes.

Given this very flexible specification of the events and policies that may be used in our system the roles of some of the event service element components may be expanded.

- The router examines the header of any incoming XML fragment and sends the information to the appropriate subsystem. Policies to the policy handler, events to the event manager and any messages that are not intended for this system are

passed to the forwarder for onward transmission via the output communications adapters.

- The policy handler examines any incoming policies and determines if they are intended to influence the operation of the event processor or event logger in which case they are routed accordingly. Or if they are intended to manage other installed software the policies are passed to the policy manager for eventual storage in the XML store.
- The policy manager interacts with the XML store loading policies, and formulating queries in response to detected events passed by the event processor.
- The event processor examines incoming events and checks if it has a policy defined to handle them. These policies will include actions such as pass to local store for a given period of time, discard as unimportant, aggregate if a large number arrive, query the policy manger for actions or in some cases pass on to another system.
- The event logger stores a local event history, this is another traffic management aid. In any system there will be a number of normal system events that only become of interest in the case of a system failure or other major problem. Rather than sending these events to a centralised store they are stored locally reducing network load. It is possible that the local store may be queried for patterns of events that might be indicative of failure conditions.
- The forwarder handles onward transmission of events, policies and any other messages received by the system.

6. Policy Store

The policy store is a key function in our system - it must be able to rapidly store and retrieve large numbers of XML fragments. In our initial prototype we will be storing a number of different types of document, events, policies and metapolicies so we require a fast and flexible XML store and search capability. In this system we are currently using XSet [14] a lightweight database for Internet applications developed at the University of California Berkeley. XSet is now in version 2 and can be described as a memory resident, hierarchically structured database with support for a partial set of ACID semantics. The version 2 release has extended the original memory resident database to include disc storage. During normal usage documents that are inserted are written immediately to disc. Insert and delete operations are logged both before and after the operation completes, the logs being buffered in memory, which is written to disc when full. XSet supports checkpoints, which may be carried out at regular intervals, this aids rapid recovery during restarts. At this time XSet doesn't support transactions, this allows it to use a coarse grain thread locking mechanism to minimise overhead and maintain high performance using a memory resident data structure allows us to gain a speed advantage during normal operation, important in a interactive environment. The number of documents stored at each element being unlike to exhaust the memory available. Currently queries in XSet are defined as XML fragments so context based searching is possible, but correct formulation of queries is complex and will be the subject of further research. Currently queries will be built in the policy manger. As previously mentioned high

performance is a pre requisite and given that we currently are flexible in our policy specification some tests were carried out on XSet using dummy files.

Number of files	Number of tags	Content	Search time on single tag msecs
2000	20	text	3.0
4000	20	text	11.0
6000	20	text	17.0

Table 2. Performance of XSet based implementation of Policy store

These results were obtained on a dual 300Mhz Pentium II PC running Windows NT version 4.0 service pack 4. This system had 256 megabytes of RAM, but use of memory was a limitation and it is highly likely that further optimisation will improve the memory usage, certainly the systems virus checking had to be disabled in order to obtain the above figures. These numbers of files seem adequate for a local store and search facility, and the performance is more than acceptable. Similar results were obtained on a single 450MHz Pentium II PC with 128mBytes of Ram running Windows 98.

Searching is only one of the performance components in XSet's operation, the other two are validation and indexing. Validation is the action of certifying the XML document against a DTD or XML Schema and indexing is the addition of a new XML document to the existing index. Indexing affects both insertions of new data into the index and recovery after a crash. Any queries requesting a document yet to be indexed will return null, so excessive indexing times may affect operation of the index. However for a production quality system with high reliability hardware the one off cost of validating and indexing a document will be far outweighed by the number of queries. Indeed in a well designed system validation and indexing may be performed in separate threads allowing uninterrupted query processing.

When used with the autonomous controller the policy store has to be able to deal with the three basic classifications of policy, guaranteed correct, errors tracked and best efforts correctness. Each of these three categories may be indicated by tags in the policy, an incoming policy will be routed to the policy manager via the policy handler. As it is inserted in the XSet store it is checked for well formedness, and validity against any supplied DTD or schema which confirms that it has an acceptable structure. This base level of operation is equivalent to the best effort correctness category of policy.

In order to achieve a higher of confidence in that the policy received is identical to the policy dispatched we must introduce further checks. At the simplest level this will just be a comparison of the structure of the document object model (DOM) tree at the originating node with the value at the receiver. This may be

achieved with a digest or hash value of the XML object; the digest is a fixed length value, typically 128 or 160 bits that represents the tree [15]. Two trees are the same if their DomHash values are the same.

As the exchange of XML documents has such a key role in B2B e-commerce there are a number of available strategies for secure Internet B2B messaging to prevent information being stolen or modified during transmission. The XML security suite [16] recently made available by IBM provides security features such as digital signature, element wise encryption and access control. This secure communication can easily be accommodated in the communication adapter, and either of these approaches would satisfy the second class of policy, that of errors tracked.

The final most stringent class of policy that of guaranteed correct is a challenge for the current implementation. XSet was designed for speed and flexibility and does not in it's current form support transactions. For our purpose there is no reason why the policy manger should not take part in a distributed transaction that ensures that the guaranteed correct class of policy are successfully inserted in the local XSet store. Using the secure communication provided for the lower policy class already offers the guarantee of correctness. The Java technology used in our prototype includes support for distributed transactions through two specifications [17] the Java Transaction API (JTA) and the Java Transaction Service (JTS). JTA is a high level, implementation independent, protocol independent API that allows applications and application servers to access transactions. JTS specifies the implementation of a Transaction Manager, which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using the Internet Inter-ORB Protocol (IIOP). Further development of our prototype will include this additional functionality.

7. Discussion

As far as we are aware existing work on policies does not consider the impact of policies being best effort, and indeed does not often recognise that best effort response is the result of most approaches to conflict resolution. Although many of the details of our approach are debatable we feel strongly that explicit recognition of the non-deterministic nature of policy based management systems will aid designers and users. We therefore hope that our work will stimulate debate and creative effort elsewhere in the management and systems community.

In the future we intend to carefully evaluate the costs and benefits of a variety of approaches to managing a large scale active services network.

8. Conclusion

A novel management architecture for large scale distributed communication systems has been described. Conflict between system policies is largely handled by removing the assumption of guaranteed execution and encouraging users to explicitly mark those policies that are critical to them using a simple grade of service matrix. These policies can then be offered a higher grade of service than other less critical policies. Initial results suggest that it would be relatively straightforward to implement the proposed architecture. A full implementation could be used to

evaluate the number of policies required in each class of service. The required system resources could then be compared with those of more conventional systems. We are confident that the proposals made in this paper offer significant advantages in terms of scalability, required resource and user satisfaction.

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] P. Cao, J. Zhang and K. Beach, "Active Cache: Caching Dynamic Contents (Objects) on the Web", Proc middleware 1998 (Ambleside).

[4] G.Parulkar *et.al* "Active Network Node Project", Washington University

[5] I.W.Marshall, C.Mallia, J.Bates, M.Spiteri, L.Velasco "Active management of multiservice networks" Proc IEE colloquium 99/147

[6] P.Mckee and I.W.Marshall "Behavioural specification using XML" Proc IEEE FTDCS '99 (Capetown), pp53-59

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

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

[9] I.W. Marshall et. al. "Application Layer Programmable Internetwork Environment", British Telecom. Technol. J., 17, 2, pp 82-94, April 1999.

[10] 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.

[11] I.W.Marshall, H.Gharib, J.Hardwicke and C.M.Roadknight, "A novel architecture for active service management" DSOM2000

[12] 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.

[13] C.M.Roadknight and I.W.Marshall, " Future Network Management - A Bacterium Inspired Solution", British Telecom Technol J. Oct 2000

[14] "XSet: A Lightweight XML Search Engine for Internet Applications", Ben Y Zhao, Anthony Joseph http://www.cs.berkeley.edu/~ravenben/xset/

[15] Digest Values for DOM (DOMHASH) Proposal, http://www.trl.ibm.co.jp/projects/xml/domhash.html

[16] XML Security Suite, http://www.alphaworks.ibm.com/tech/xmlsecuritysuite

[17] TRANSACTIONS AND JAVA™ TECHNOLOGY, http://java.sun.com/j2ee/transactions.html