# Using Active Processes as the Basis for an Integrated Distributed Network Management Architecture

Dominic P A Greenwood[1] and Damianos Gavalas[2]

[1] Fujitsu Telecommunications Europe Ltd.
Northgate House, St. Peters Street, CO1 1HH, Colchester, UK
Tel: +44 (0)1206 363002
D.Greenwood@ftel.co.uk
[2] Communications Networks Research Group
Electronic Systems Engineering Department, University of Essex
CO4 3SQ, Colchester, UK
Tel: +44 (0)1206 872425
dgaval@essex.ac.uk

**Abstract.** Active, mobile processes or agents are a technology whose features promise to have a significant impact on the distributed management of communications networks. Through the interaction of agents with one another and their device-based environments, active network architectures arise. This paper outlines a modular architecture that exploits these features to create a network-element management framework supporting both agents and conventional SNMP messaging. A process kernel is described which houses both pre-loaded and visiting agent processes, the pre-loaded processes having well defined functions including automated device registration and local fault diagnosis. An operational sequence is discussed, highlighting the beneficial usage of agents for certain automated tasks and an experiment measuring the response time of agents versus centralised SNMP polling is evaluated.

**Keywords:** Active networks, distributed network device management, agents, convergence, SNMP, programmatic fragmentation.

## 1 Introduction

With the current trend towards the seamless integration of heterogeneous telecommunications and data networks with vendor specific devices, the requirement for flexible and comprehensive management architectures is substantial. As the IP protocol begins to take a foothold in the telecommunications world, due to its low cost and ease of integration with local private networks, it is more than likely that a proliferation in the number of addressable network elements is imminent. From such

Stefan Covaci (Ed.): IWAN'99, LNCS 1653, pp. 199-212, 1999.

growth arises the inevitable requirement for end-to-end QoS, that must be managed throughout a packet's entire network path for guaranteed high-QoS transport. In addition, the user driven demand for rapid evolution in service availability will ultimately lead to user-customisable service level behaviour, which in turn will drive the deployment of new services. All this leads to one inescapable fact; that current management architectures and protocols will not be able to scale to the required degree of complexity.

The current telecommunications management stack is commonly viewed as three primary layers; the device/element layer, the network layer and the service layer. The latter of these tends to operate as a superimposed market above the former two, which remain ingrained with simple message based protocols such as SNMP [1] and CMIP [2]. Although these are widely accepted standards, their limitations become evident when compared with a hybrid of message and active process management environments, as shown by [3] and [4], and as presented in this paper. The traditional message based communication mechanism used by both SNMP and CMIP relies heavily on frequent polling of devices by a central management station, which analyses all incoming data searching for problems and changes in condition. In by far the majority of cases, bandwidth is consumed to learn nothing other than that the network is operating within acceptable parametrical boundaries.

The argument against moving towards more distributed and active management architectures assumes that current networked devices have an adequate level of control. However, this requires that network administration remains a centralised, operator-dependent and message based process, ignoring the many benefits of having devices capable of autonomous management and configuration. Even if we do not consider the benefits of a seamless, element oriented management stack, there still remains the potential for improvement over traditional SNMP/CMIP as shown in [5], which discusses the use of active processes for performance management.

Many of these issues are highlighted in [6], which discusses the utility of programmatic elements in the upper layers of the management stack, i.e. service and customer management (a superset of service management). At the more fundamental level of network element management, [3] and [4] consider the implications of migrating network management towards progressively more active architectures.

In light of these factors and the extensive work conducted to date within the field of active networks, this paper proposes a modular architecture based on active processes, which adopt specific operational functions within a managed device.

Section 2 outlines the agent process and highlights those features applicable to this topic. Section 3 introduces the modular, network element management architecture and discusses its components. Section 4 describes several operational sequences that the architecture may undergo subsequent to activation and presents some preliminary results comparing the use of agents to centralised SNMP polling for the calculation of an aggregated function. Finally, Section 5 draws conclusions on the implications of the proposed technology.

## 2 The Active Process Component

The mobile programmatic element, or *agent* for the purposes of this paper, has existed for many decades, initially in the form of computer viruses. In essence an autonomous entity, the agent interacts with its environment and other active processes to achieve an overall goal. Contemporary features have led to the agent finding a role in areas associated with distributed computing [7] and, by association, with inter-communicating networked devices. For the purposes of network device management, as discussed in this paper, the agent takes the abstract form described by Figure 1:

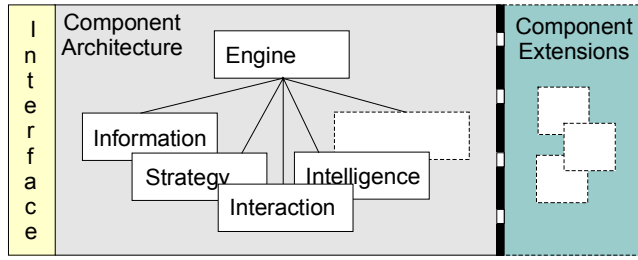


**Figure 1.** Abstract component-based structure of a generic process agent

The agent consists of a set of core functional components linked to the process engine. The interaction component controls the agent's interface to other agents and the network environment. In this model tautology, only specific query and transaction primitives are defined, allowing the agent to confer with its local host regarding resource manipulation and device configuration. Additionally, semantic language constructs allow tiered interaction with peer agents, the level and complexity of communication relating to intrinsic capabilities of the agent through its intelligence and strategy components, and the specific management task. To clarify, agents may require the ability to negotiate for resource access, perhaps acting as a broker on behalf of service provider. As such, the agent can barter, which in turn requires a concept of credit exchange. The FIPA specified ACL [8], defines the syntactical and semantic clauses required for such interaction, and is the subject of current developmental work.

The information component of Figure 1 defines any knowledge that the agent has concerning its specific task, and acts as the logical repository for any conveyed management information.

Also shown is a facility to integrate component extensions into agent profiles. Achieved through the process of programmatic fragmentation, an agent may access the additional routines and data segments through the equivalent of dynamically linked object instantiation. This technology is related to the dynamic code integration described in [9], for the process oriented APRIL language, but implemented in Java for this architecture. Java has been chosen primarily due to its conceptual ethos as a portable, strongly typed language. Although there are problems currently with Java's processor intensive support environments, it remains entirely suitable for an experimental framework.

## 3 Process Oriented Device Management

Having briefly outlined the proposed architecture and discussed characteristics of the process agents' employed within it, we now discuss the integrated device architecture itself. Shown in Figure 2, the proposed system is separated into two primary sections: the Device Resources and Device Kernel, and the Process Kernel. Between these two lies the Protocol Interface and to the right of the diagram, the External Process Filter Interface (EPFI).

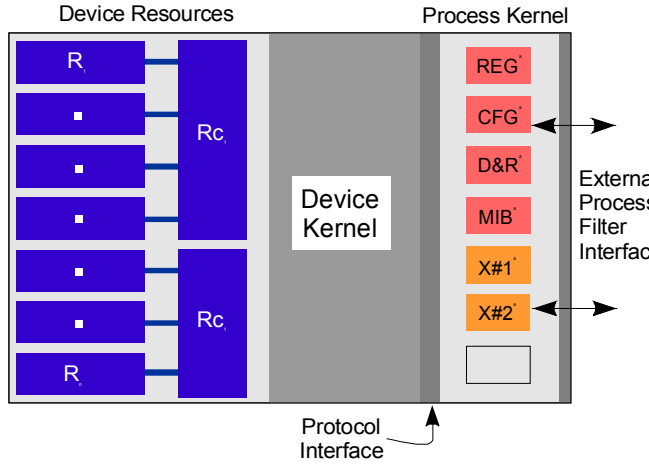


**Figure 2.** Agent enabled device management architecture

Within the Device Resources section is found an arbitrary resource configuration, dependent upon the device function. In general, $Rc_k$ are consolidation units, with $R_1$ to $R_n$ representing individual resource components. The Device Kernel is the local control processor specific to the type of device, responsible for hardware management and controlling internal packet transfers and flows.

The Process Kernel section resides logically adjacent to the Device Kernel separated by the Protocol Interface. A Java virtual environment, the Process Kernel allows agent processes to execute as threads, accessing the device resources via the device kernel and communicating locally with one another. In the proposed architecture, each active device is preinstalled with four agent processes: the $REG^a$, the $CFG^a$, the $D\&R^a$ and the $MIB^a$. These represent dedicated task agents for Registration, Configuration, Diagnose and Repair, and MIB Messaging respectively. The $X\#n^a$ agents represent non-specific process agents that may arrive from an external source, or be generated internally by an Agent Generator process, which is described later.

The following section now describes each of these core components in detail.

## 3.1 The Protocol Interface (PI)

The Protocol Interface forms a security shield and protocol translation boundary between the primary device components and the management process kernel. In its capacity as a security shield, all queries and configuration commands originating from the process kernel are authenticated to ensure validity. Additional protection could potentially include data encryption, although this is unlikely to be feasible when considering real-time operating constraints. It is more conceivable that a trust relationship would exist across the interface, local to the device.

The protocol translation function of the PI enables the managed device to inter-operate with proprietary agents, whether they have common or proprietary interfaces. The aim of standardisation bodies, such as FIPA [10], is to ensure that mobile agents have a common interface with an accepted baseline communication protocol. It may however often be the case that a vendor requires restrictive control over device or agent access by defining a proprietary protocol. This ability, whilst reducing inter-operability, is potentially a flexible and useful control of agent access to the device kernel and resources. Figure 3 illustrates a selection of interface configurations:

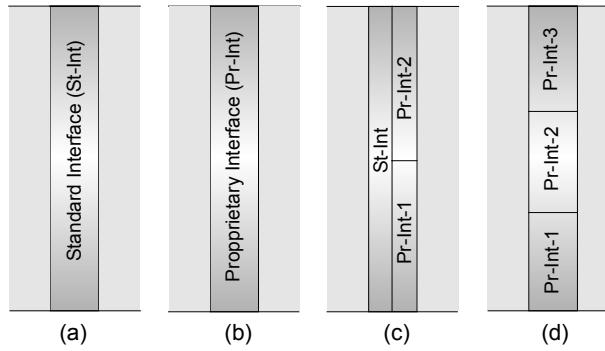


**Figure 3.** Protocol Interface, optional protocol configurations.

As can be seen, subject to requirements the PI may take any of the configurations shown in Figure 3. The most flexible of these is that shown in (d), where the underlying common interface has a number of optional proprietary protocol overlays.

## 3.2 The External Process Interface (EPFI)

Not yet implemented, this component of the proposed architecture is an optional external interface that provides a protective barrier between external network devices and the process kernel, to guard against corrupt or malicious code access. Whereas the PI is responsible for verifying individual resource access and control signals, the EPFI validates agent authenticity against accepted criteria, before passage into the process kernel. This feature must remain optional due to the proprietary nature of many devices; it may not be possible to define widely accepted, trusted and reliable authentication procedures.

## 3.3 The Process Kernel

The Process Kernel is the host environment of the device management system and is specifically responsible for controlling the flow of information between resident agent processes and managing its internal and external interfaces. In the current implementation, the kernel is a Java Virtual Machine with a native interface to the Device Kernel. Each resident agent process is coupled to the system via a thread link from where it may signal other agents within a transaction forum.

A fundamental feature of agents is that they may be functionally extended at run-time by using either dynamic libraries or language dependent extension features. This implies that functionality may be changed 'on-the-fly' without the need to replace the entire agent – although this may be necessary in certain cases.

The essential agents included in all managed devices, as shown in Figure 2, are the Registration agent (REG[a]), the Configuration agent (CFG[a]), the Diagnostic and Repair agent (D&R[a]) and the MIB Messaging agent (MIB[a]). These agents may be suspended, terminated, replaced or updated according to requirement and correct authority.

As previously noted, the PI performs any necessary protocol translation and security authentication before transactions may take place between active agents.

### 3.3.1 The Registration Agent (REG[a])

The REG[a] is a replaceable agent process, activated at device initialisation (boot-time), and responsible for registering its host device with other active network devices. The REG[a] must contain a unique identifier (commonly a MAC or IP address) for the new device or DHCP [10] server location. Using a local node discovery algorithm, such as the IPv6 anycast function [11], the REG[a] locates a nearest neighbour and migrates to its docking facility[1] via a recognised TCP/IP port.

The REG[a] registers its host device with the device visited and, if specified, exchanges resource information. Ideally, this exchange would allow the REG[a] to identify the visited device's resource function and availability, routing tables and particularly, the location of any local management stations. Naturally though, such information may be unavailable or restricted, but regardless, the REG[a] will continue to visit local devices until its terminate/return criteria have been satisfied.

Dependent upon prevailing network conditions, or pre-specified behaviour, the REG[a] either messages or collates. In message mode, information is returned to the source device from each device visited, before termination criteria are reached. In collation mode, information is accumulated by the REG[a] process, before returning to its host. Evidently, a trade-off exists between assured arrival of information and bandwidth preservation.

---

[1] The IPv6 'anycast' function searches for *any* available node within a specified domain member set.

### 3.3.2 The Configuration Agent (CFG[a])

The CFG[a] AGENT controls device self-configuration, primarily through patterned calibration of device parameters. A 'pattern' being an array of pre-determined parameter settings defining the configuration of both resource and management subsystems, which may be applied by the CFG[a] in response to operating conditions. The level of autonomous control varies from complete pattern selection to pattern parameter modification. Considering the device architecture in Figure 2, CFG[a] configuration patterns control such features as:

Device status.
Resource configuration and access control.
Thread and socket availability and prioritisation.
Agent design, strategies and production.
Security, authentication and encryption.

When the network device is installed, the CFG[a] specifies initial setup conditions. These may be subject to change according to information returned by the REG[a] process. For example, a domain management controller may wish to limit the production of X#n[a] agents according to specified criteria. The device may be forced to adopt this policy and hence update its internal configuration.

### 3.3.3 The Diagnose / Repair Agent (D&R[a])

The D&R[a] module provides an internal service that monitors the behaviour of device resources and management components, watching particularly for faults and anomalies. A diagnosis may include the identification of an imminent resource failure based on observed behaviour. Such a situation would cause the repair mechanism to switch to a spare resource, if available, and notify registered management stations of the situation.

The specific processing capabilities of the D&R[a] are device and vendor dependent, and hence the diagnosis function may range from a fault location report to a complex diagnosis of the fault and potential causes. In the latter case, processing may include the ability to draw hypotheses based on known facts and reasoned heuristics, enabling the device to draw its own conclusions and commence a repair programme.

### 3.3.4 The Messaging Agent (MIB[a])

In order that the framework be compliant with SNMP and CMIP standards, the MIB[a] provides a standard Management Information Base and a query interface for basic messaging. For the purposes of many management enquiries and actions, a simple message service is appropriate by comparison with a relatively heavyweight mobile agent.

### 3.3.5 Non-Specific Function Agents (X#n[a])

In addition to the system agents, the management subsystem may create and host dynamic agents (X#n). These may either be created internally by the AGen (see Section 3.4) or arrive from an external source, via the EPFI. Before activation, an X#n[a] is assigned a process thread by the kernel that when active, forms a conduit through which the agent may communicate with other resident agents and query managed resources. Some examples of the type of task an X#n[a] will conduct are:

Interrogate device resource availability.
Negotiate for resource access.
Supply code updates for resident agents.
Deliver active processes to update the host device functionality.
Provide a legacy management interface for SNMP/CMIP compliance.

The interface of any agent that arrives from an external source must conform to security and protocol requirements imposed by the PI. Otherwise the agent will be terminated and its source node notified if required.

## 3.4 The Agent Generator (AGEN)

This component is optional, dependent on whether the vendor requires the device be capable of creating its own agents. If resident, the AGen exists as a separate process within the process kernel, although for many managed devices it is normally sufficient that they accept X#n[a]'s from external sources, without taking an active role in agent generation. The absence of an AGen component restricts the creation of agents within the managed network, thereby regulating their proliferation. If required however, an AGen process may be uploaded from an external source subject to device authentication and publishing rights.

The AGen creates an agent according to blueprint patterns it holds. Each describes in detail the framework, components and interfaces required in building the desired agent. If any components are unavailable at the local host, a message is dispatched to other known devices (including management stations and vendor sites) to request copies of the required code. This naturally creates a distributed component library where core elements are present at all devices that require them, and more specialised ones located at known repository sites.

Agents may take on a variety of tasks ranging from basic SNMP type operations to complex autonomous configuration and service management. The AGen must therefore have access to multiple component types, and be sufficiently complex to construct information channels between them.

# 4 Experimental Network Architecture

The development of an experimental architecture for evaluating the concepts described here is in progress. Based on the generic device construct described by Figure 4, the hardware subsystems including the Device Kernel and Device Resources are currently abstracted into the software simulation framework. Given that several of these simulated devices may be interconnected forming a network, a traffic stream is introduced emanating from an external source. Each actively managed device then filters the packet flow, searching for agent and message signatures in their headers sending normal traffic onto the resource and directing agent and message packet sequences into the Process Kernel for re-assembly.

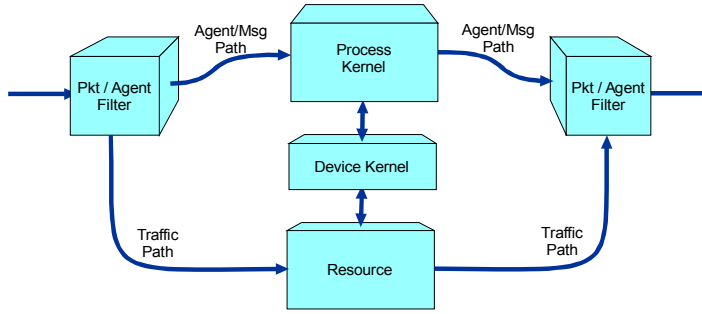


**Figure 4.** A generic device

## 4.1 An Operational Sequence

To illustrate the proposed architecture in operation, consider an abstract network configuration as described by Figure 5:

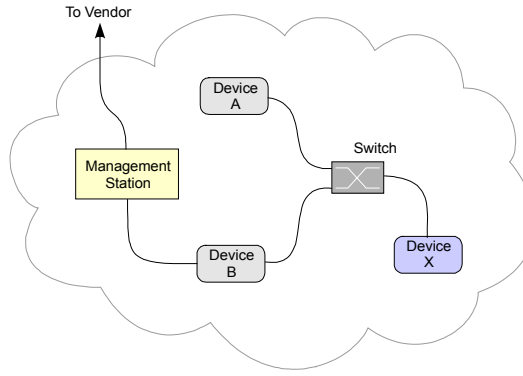


**Figure 5.** Simple network configuration model

Here two existing devices, DeviceA and DeviceB are connected to a switch, with DeviceB also connected to a local Management Station (MS). At an arbitrary point, a new device, DeviceX, is introduced to the network and given a direct path to the switch. The timing diagram shown in Figure 6 describes the interaction between these devices shortly after DeviceX is introduced and activated. The Agent and Msg tags indicate the transmission of an agent or standard message, respectively.

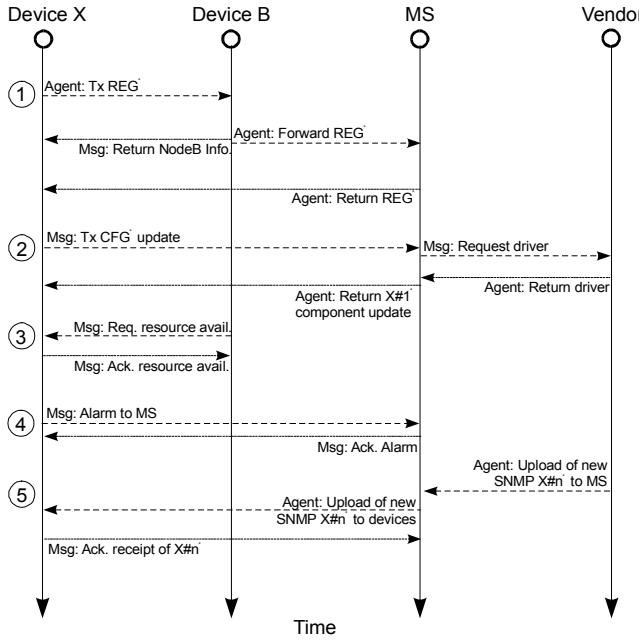


**Figure 6.** Interaction timing diagram

Following is a description of each of the listed event sequences, numbered 1 to 5:

① DEVICE REGISTRATION

Following activation of Device X, the device's REG[a] is emitted on an *anycast* towards the first detectable device. In this sequence, the first located node is Device B, which accepts the REG[a] and forwards it onto the known regional MS. In addition, the REG[a] requests of Device B (and any additional intermediate nodes before an MS is reached), that it returns a message to the source device to confirm arrival.

Once the MS receives the REG[a], it registers the new device and notes its available resources. All known devices (except Device B) are then informed of the presence of the new device. The REG[a] then returns to its source, with the inclusion of MS management information relating to resource location, resource and service availability, and network device addresses.

② CFG[a] UPDATE
This sequence illustrates that during device configuration, a resource is detected for which no driver is locally available. The MIB[a] therefore sends a message to the local MS requesting download of a extension component for the resident CFG[a]. The MS in turn resolves the vendor location for the required driver and requests the update be sent as an installable X#n[a] agent, which is then transferred by return to the requesting device.

③ RESOURCE AVAILABILITY QUERY
Under normal operational conditions, the device will regularly receive requests for management information or resource availability. This sequence illustrates a device requesting resource information from Device X.

④ FAULT DETECTION
This sequence illustrates polling of the MS with an alarm notification; a resource fault detection at the new device. The fault is initially detected by the D&R[a] system monitor, which attempts to diagnose the fault and if no local repair is possible, inform the MS. The MS in turn acknowledges alarm receipt, attempts to resolve alternative resources for affected services and takes appropriate action to repair the device.

⑤ X#n[a] UPLOAD
The final sequence describes the situation where a vendor releases an updated version of an X#n[a] that certain devices utilise. The replacement or update code is first sent by the vendor to all registered MS's from where it is dispatched to all affected devices as an agent X#n[a]. Each device returns a receipt message to the MS upon successful installation.

The modular nature of the architecture allows any components to be updated or replaced as required, thus providing additional functionality alongside flexibility and reliability.

## 4.2 Example of an SNMP Management Task

In order to test the potential improvement over traditional SNMP centralised polling, this section presents some preliminary results of an agent based calculation of an aggregated MIB function. Otherwise known as a health function, the aggregation consists of multiple MIB variables representing the percentage of discarded IP input datagrams over the total number of datagrams received during a specified time interval. When using a traditional centralised polling scheme the value of this function would be computed by the manager upon return of all object values from the polled static management agents. Hence, if $S_r$ is the average request/response size, and polling of $n$ devices for $v$ operational variables is applied, the wasted bandwidth ($BW$) for $i$ polling intervals would be:

$$BW_{standard} = 2 * S_r * n * v * i \quad (1)$$

Alternatively, a sequential agent based scheme may be applied, where a single agent travels from the original source, visiting each device in turn before returning the calculated results from the last polled device. The health function calculation is performed at the local agent host, leading to a balanced distribution of the computational burden. Additionally, considerable compression of data is achieved as a large number of observed operational variables are reduced to a single value. Thus, assuming an agent average code size of $A_c$ and state information size $A_s$, the resulting overhead would be:

$$BW_{agent} = (A_c * n) + (A_s * n * i) \quad (2)$$

Here the $v$ variables are aggregated into one. The first term of equation (2) describes the overhead imposed when broadcasting the agent code to all management servers, whilst the second represents the bandwidth consumed by the agent state transfers between the manager and the polled devices. Thus, for large $v$ and $i$, agent based polling is clearly less bandwidth intensive than centralised polling.

In Figure 7, the agent based polling scheme is compared to standard SNMP. The response time for the acquisition of the specified health function is measured, with Figure 7a showing that the results of applying a single agent to the task does not scale well as the number of devices increases. If however an agent is assigned a fixed number of devices, thereby partitioning the network into logical segments, is should be possible to improve the response times.

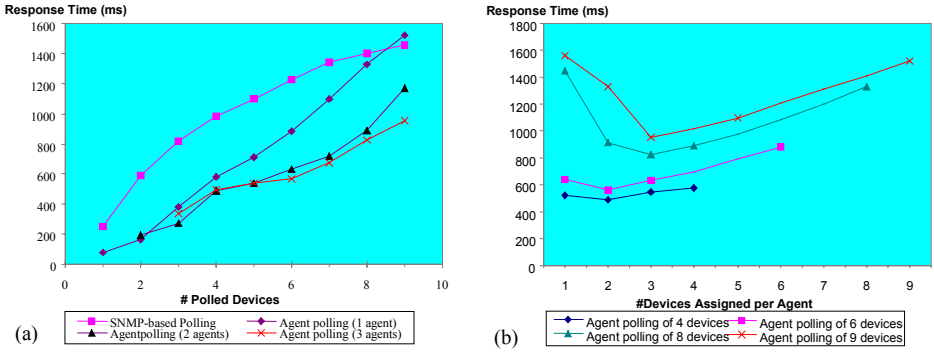


**Figure 7.** Response time (a) Number polled devices, (b) Number of devices assigned per agent.

Figure 7b shows the results of applying such partitioning. Depending on the managed network size, the optimum number of segments may be determined from the minimum point of the corresponding curve. For instance, in a network of six devices, the response time is minimised when each agent is assigned to two devices, i.e. the network is segmented into three domains. Hence, any active domain manager should autonomously match the number of segments to the number of required polling agents, according to the volume of managed devices. This issue is currently under investigation.

## 5 Conclusions

The adoption of management architectures as that proposed in this paper is dependent on device vendors integrating process execution environments into their devices. When such apparatus becomes readily available, the implication is that active network architectures will begin to evolve further. This paper has proposed a modular, process based management architecture that highlights the benefits of active code agents, including flexibility, reliability and inherent scalability, all core issues in the convergence of the telecommunications and data-communications worlds.

Preliminary results have also been described for an experiment highlighting the beneficial use of agent processes for simple aggregated SNMP function calculation. Response time was shown to have improved over that for traditional centralised polling, with an intrinsic reduction in bandwidth consumption also apparent due to the elimination of broadcast polling.

Current work pertains to further implementation of the described architecture and related issues concerning the function and interaction of agent processes.

## References

[1] Case J., Fedor M., Schoffstall M., Davin J., 'A Simple Network Management Protocol (SNMP)', RFC 1157, DDN Network Information Center, SRI International (1990).

[2] ISO/IEC 9596, Information Technology, Open Systems Interconnection, Common Management Information Protocol (CMIP) – Part 1: Specification, Geneva, Switzerland (1991).

[3] Susilo G., Bieszczad A. and Pagurek B., 'Infrastructure for Advanced Network Management based on Mobile Code', Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98), New Orleans (1998) 322-333.

[4] Ku H., Luderer G., Subbiah B., 'An Intelligent Mobile Agent Framework for Distributed Network Management', Proc. IEEE Globecom'97, Phoenix AZ (1997) 160-164.

[5] Gavalas D., Greenwood D., Ghanbari M., O'Mahony M., 'An Infrastructure for Distributed and Dynamic Network Management based on Mobile Agent Technology', IEEE Int. Conference on Communications (ICC'99), Vancouver (1999).

[6] Magedanz, T., Rothermel, K., Karuse, S., 'Intelligent Agents: An Emerging technology for Next Generation Telecommunications ?', Infocom'96, San Francisco, CA (1996).

[7] Carzaniga, A., Picco, G. and Vigna, G., 'Designing Distributed Applications with Mobile Code Paradigms', Proc. 19th Int. Conf. On Software Engineering (1997) 22-32.

[8] FIPA 97 specification – Part 2 : Agent communication language, http://www.fipa.org/spec/FIPA97.html

[9] McCabe, F. G. and Clark, K. L., 'APRIL: Agent Process Interaction Language.', Intelligent Agents, Lecture Notes in Artificial Intelligence, 890, Wooldridge, M. J. and Jennings, N. R., Eds, Springer-Verlag (1995) 324—340.
[10] Chiariglione, L. 'FIPA: A General Overview', http://drogo.cselt.stet.it/fipa/ - Papers
[11] Dynamic Host Configuration Protocol: Draft Standard RFC 2131: IETF (1997), ftp://ftp.isi.edu/in-notes/rfc2131.txt
[12] Narten, T., Nordmark, E., Simpson, W.A., 'Neighbor Discovery for IP Version 6 (IPv6)', IETF, http://www.ietf.org/internet-drafts/draft-ietf-ipngwg-discovery-v2-03.txt