Network Monitoring with Asynchronous Notifications in Web Service Environments

Torsten Klie[1], Florian Müller[2], and Stefan Fischer[3]

[1] Technische Universität Braunschweig
Institute of Operating Systems and Computer Networks
Mühlenpfordtstr. 23, 38106 Braunschweig, Germany
tklie@ibr.cs.tu-bs.de
[2] Universität Düsseldorf
Institute of Computer Science / OS Department
Universitätsstr.1, 40225 Düsseldorf, Germany
marc-florian.mueller@uni-duesseldorf.de
[3] Universität zu Lübeck
Institute of Telematics
Ratzeburger Allee 160, 23538 Lübeck, Germany
fischer@itm.uni-luebeck.de

Abstract. With the ongoing growth of the Internet, a reliable communications infrastructure becomes more and more important. To monitor large networks, asynchronous notifications are needed to inform management systems about critical events. The SNMP framework provides Traps which have been used for monitoring successfully for many years. However, due to shortcomings of SNMP, XML and Web services get more and more attention from the network management community. If the new technologies should replace the existing ones, they must also provide asynchronous notifications. In this paper, we describe notification solutions in traditional as well as in new management frameworks. Moreover, we compare them with a simple approach based on Web services we have implemented.

1 Introduction

With the ongoing growth of the Internet, more and more people and enterprises depend on a reliable communications infrastructure. Therefore, management systems that can deal with the complexity of today's networks are needed. To monitor large scale networks, asynchronous notifications, which inform a management system about events such as failures and state changes, are helpful, since it is often not desired to poll the state of a large number of devices regularly if the states only change once in a while; valuable network resources and – probably even more important – computing power of the manager would be wasted.

The SNMP Framework [1] is the standard management framework by the IETF. Since SNMPv1, asynchronous notifications (called Traps here) are available. SNMP is a management specific technology, which makes it difficult and

expensive to find skilled personnel. XML-based technologies such as Web services on the other hand, are a technology that is independent from the application domain, so experts in that field are easier to find. Several frameworks for Web services based management have been developed (e.g. OASIS WSDM [2] and DMTF WS-Management [3]). Furthermore, the IETF has standardized NETCONF [4], an XML based protocol for configuring network devices. Syslog [5] is a traditional notification protocol that is independent from management frameworks.

In this paper, we will describe the basic principles of asynchronous notifications in current management frameworks. Furthermore, we will present our own notification prototype. The paper is structured as follows. In Section 2, we will show, how notifications are used in traditional approaches. Section 3 will then describe notifications in newer approaches. In Section 4, we will describe our own prototype implementation. We will end the paper with a conclusion and an outlook on further work.

2 Traditional Approaches

2.1 SNMP Framework

SNMP [1] has been widely used since its first version (SNMPv1) was standardized in 1990 and it is still in use today. The main focus of SNMP lies on hardware devices such as routers, switches, bridges, DSL modems, etc., although it is also possible to manage software systems such as Web servers, database servers, etc. The fundamental design idea was simplicity. SNMP poses only minimal resource requirements to the devices. Therefore, SNMP's design and operations are kept simple.

The SNMP model follows the manager-agent paradigm. The manager initiates the communication using a `GET` or a `SET` operation, to which the agent replies with a `GET-RESPONSE` or a `SET-RESPONSE`. SNMP works on managed objects that are defined in a Management Information Base (MIB). The data models represented by a MIB are written in the Structure of Management Information (SMI) language [6], which uses an adapted subset of ASN.1. By July 2004, there have been more than 200 IETF Standard MIB modules defined by different IETF working groups and more than 600 vendor specific MIBs [7].

In some situations, it is desired to initiate the communication the other way round: the agent notifies the manager about an event that has happened (a critical situation, a reboot, etc.). In SNMPv1, the agent can send `TRAP` messages to managers. Since SNMPv2, `NOTIFICATION`s are a generalization of unconfirmed `TRAP`s. They also include confirmed `INFORM`s, which were introduced in SNMPv2.

An SNMPv1 `TRAP` PDU consists of a PDU type marker indicating "`TRAP`", an identifier of the originating network management subsystem, the IP address of the originating agent, the `TRAP` type (`coldStart`, `warmStart`, `linkDown`, `linkUp`, `authenticationFailure`, `egpNeighborLoss`, or `enterpriseSpecific`), a code that denounces more specific information, a time-stamp, and additional information such as variable bindings. In SNMPv2, all information (except the PDU

type marker) are part of the variable bindings. The time-stamp has been replaced with the uptime of the agent, the IP address has been removed.

2.2 Syslog

The BSD syslog protocol [5] is a simple notification protocol, which is very easy to implement. It is a stand-alone protocol, that means it is not integrated into any management protocol framework. To keep the protocol simple, no assumption is made about the formatting of a message. Devices supporting syslog must be configured to display a message (e.g. writing a log file) or to forward it to another system (or both). Each process on every device is supposed to create notifications. UDP (port 514) is used as the transport protocol.

Message categories are called facilities. There are 24 facilities, starting from 0: kernel message, over 1: user-level messages, 2: mail system, 7: network news subsystem, and 9: clock daemon to facilities that are reserved for local use (16: local use 0 to 23: local use 7). Moreover, there are 8 levels of severity (from 0: Emergency to 7: Debug).

A syslog message consist of three parts: A priority value (PRI), a header (with time-stamp and hostname), and the message body. The PRI value is calculated using the following formula: $PRI = 8 \times \text{facility code} + \text{severity}$. The time-stamp and the hostname are optional but its use is recommended. If the originating system does not know its hostname, the IP address can be used instead. The message part consists of a TAG (free-form string of max. 32 alphanumeric characters giving details about the event) and the contents of the message. Due to length restrictions (the total length of the syslog packet must not exceed 1024 bytes) the payload may be truncated.

Syslog supports forwarding of messages using relays. A relay can forward the message to other relays.

Standardization efforts for syslog extensions that are secure and reliable have to deal with the lack of a standards-track and transport independent RFC. Therefore, the IETF proposes a new layered specification [8], which has the following main differences:

- The new specification is transport independent. However, all implementations must support at least the transport over UDP.
- The time-stamp format offers more precision and complies with RFC 3339[9].
- The format of the hostname is more specific.
- APP-Name, PROCID, and MSGID have been added to the header (as a replacement for TAG.
- Structured data fields can be used between header and message body. Structured data, which contains an ID, the parameter name, and the parameter value, can be added between header and body.
- The behavior of relays is not specified.
- The minimal message size a receiver must support has been reduced to 480 octets, although implementations should support messages up to 2048 octets. More octets can be used, if needed.

3 New Management Approaches

3.1 DMTF WS-Management

WS-Management [3] is a SOAP-based protocol that is intended for configuring and monitoring systems such as computers (PCs and servers) and applications (Web services, databases, etc.). It provides a core set of management operations: find resources and navigate between them (DISCOVER), read and modify individual management resources (GET, PUT, CREATE, RENAME, DELETE), view the contents of large collections (ENUMERATE), enable the reception of events (SUBSCRIBE), and perform specific management functions (EXECUTE). Its specification poses constraints to Web services protocols and formats such that the implementation of the Web services for management will not have a large footprint on managed resources. These Web services should be composable with Web services of other specifications. As a transport protocol, only HTTP 1.1 and HTTPS are available for interoperability reasons.

WS-Management makes intensive use of other "WS-*" specifications. Managed resources are addressed using WS-Addressing endpoint references. For access, `Get`, `Put`, `Create`, and `Delete` from WS-Transfer are used. WS-Enumeration is used for multi-instanced resources. WS-Management uses WS-Eventing for notification purposes (see Section 3.2).

3.2 WS-Eventing

The draft specification [10] provides a framework for notifications in a Web services based environment. Wherever possible, other Web services specific specifications ("WS-*") are integrated. The goal of the specification is to enable a secure, reliable, and/or transaction based delivery of notification messages.

Message delivery is subscription based. One of the most important assumptions is the expiration of these subscriptions. Furthermore, besides an asynchronous, unconfirmed delivery (called Push Mode), the framework supports other delivery modes (although the specification deals only with Push Modes). In general, a delivery mode can be seen as an abstract mechanism that enables event sources end event receivers to freely negotiate transport session parameters that fit their specific requirements.

Subscriptions can, but do not have to be managed by the event source. The event source may delegate subscription requests to an external entity called subscription manager.

3.3 OASIS Web Services Distributed Management

The OASIS Web Services Distributed Management (WSDM) standard [2] tries to unify management infrastructures by offering a framework that is independent from vendors, platforms, network types, and protocols. It is intended to enable management for a great variety devices from network devices such as routers to consumer devices such as DVD players, including Web services as well. Therefore,

it provides two sets of specifications: Management Using Web Services (MUWS) and Management of Web Services (MOWS).

Manageable resources, addressed with WS-Addressing endpoint references, are in the main focus of WSDM. Managers (called "manageability consumers") can retrieve management information from a resource, either by asking explicitly (`GetResourceProperty` and `GetMultipleResourceProperty`), or by subscribing to certain events of a resource (`Subscribe`, `PauseSubscription`, and `ResumeSubscription`). What kind of management functions a resource supports (i.e. its "manageability capabilities") is specified in the resource property document that is referenced from a WSDL port type. The manager can retrieve portions of it using `QueryResourceProperties` and a filter (such as XPath). To modify management information, `SetResourceProperties` is used.

According to WS-BaseNotification [11], there are several ways for a manager to receive notifications. If it provides a `Notify` operation, the manageable resource will use it to submit the notification. Otherwise, the manager can ask the resource using `GetCurrentMessage` for the last notification belonging to a given topic.

3.4 IETF NETCONF

The NETCONF protocol defines operations for managing network devices. Configuration data can be uploaded, downloaded, and manipulated either as a full or a partial data set. This can be done using a formal API exposed by the managed device, which is based on an RPC encoded in XML.NETCONF can use a large number of application protocols for transport. Currently, there are RFCs that specify bindings for SSH, SOAP, and BEEP.

NETCONF distinguishes between state data and configuration data. Configuration data can be retrieved with `<get-config>` and modified with the operations `<edit-config>`, `<copy-config>`, and `<delete-config>`. `<get>` retrieves all available data.

The NETCONF protocol can handle several data stores, i.e. sets of configuration data. The active data store is called `<running>`. In addition, there can be a `<candidate>` and a `<startup>` data store.

NETCONF exploits the tree structure of the data that is shipped with a `<rpc-reply>` message from an agent by allowing to select parts of the query result by means of XML. This subtree filtering allows the manager to specify a filter that selects only the nodes the manager is interested in. The filtering will be done on the agent, thus saving bandwidth. The mechanism works with the XML tree structure, that means the agent can perform the filtering without dealing with any data model specific semantics.

In order to allow supplementing the base NETCONF specification with additional sets of functionality, NETCONF supports the specification of capabilities. Capabilities are augmentations to the basic NETCONF operations and the content of these operations, respectively, and are identified by a uniform resource identifier (URI).

The most important requirements that NETCONF notifications [12] should fulfill are reliable message transport, subscriptions, filter mechanisms, and a reasonable message size limit. NETCONF notifications should be run as a subsystem on NETCONF agents (called central event processor). If an event occurs in the system components, the central event processor will use a notification logging system to store the event details. Furthermore, it will forward the notification to event streams. The NETCONF server can receive certain streams and forward them to subscribed NETCONF clients.

Managers can subscribe to event streams of an agent that supports the notification capability using the `<create-subscription>` operation. Subscriptions can be modified and cancelled with the operations `<modify-subscription>` and `<cancel-subscription>`, respectively. Managers can receive a list of available event streams using `<get>` with `<eventStreams>` as subtree filter. Notifications are usually sent as "one-way messages", thus every notification message is a well-formed XML document, which can be processed directly after receiving it, without having to wait for further events.

The manager is probably not interested in all messages from an agent. To restrict the messages to the desired ones, filter rules can be used. An important criterion is event class[1]. Moreover, filtering can be done on the data model using the NETCONF filter mechanisms (subtree and XPath).

4 Our Approach

4.1 YAMA - Yet Another Monitoring Approach?

In the previous sections, we have described several approaches for asynchronous event notification for monitoring purposes. Why are we introducing our own approach? Except for syslog, all other approaches are embedded into a (management) framework. We wanted to create a monitoring solution that is independent from existing frameworks. To use the approach in our Web services based management environment [14], support for Web services was required.

In the following, we will describe the design of our approach, reveal some implementation details, and present our initial evaluation results. Furthermore, we will compare our approach to the approaches discussed in the previous sections.

4.2 Design

Our approach follows the manager-agent paradigm. In order to receive notifications from a specific agent, a manager must subscribe to certain events. Agents are connected to the managed objects as in the SNMP framework: they could be integrated into the device or system, or they could work as a proxy agent outside the system.

[1] Earlier versions of the draft (such as [13]) included a list of event classes, but they have been removed since they were considered data model specific.

Before an agent starts to send notifications to a manager, it must know which manager is interested in what kind of information. Therefore, managers must choose agents, managed objects, and conditions that have to be fulfilled before a notification is sent. The reason for this is twofold. Firstly, the system should be flexible w.r.t. the type of notifications that are sent. Secondly, network resources should not be wasted by unwanted messages. If a manager wants to receive notifications from an agent, it sends a configuration request to that agent specifying the chosen object and the conditions. Hereby, the manager delegates the monitoring to the agent. For example, the manager could specify "processor load" as the object and "greater than 3.0" as the condition. The manager then does not have to perform any further monitoring, this is done by the agent. If the processor load exceeds 3.0, the manager will receive a notification from the agent.

Objects that can be monitored this way are called monitoring functions. There are two kinds of monitoring functions: functions that collect and aggregate information about the resource (statistical monitoring functions) and functions that monitor a resource for certain events (event monitoring functions). An event is the occurrence of a certain state at the agent. The events can be faults, state changes, reached thresholds, external input, etc. Events can be tagged with an arbitrary number of labels. Currently, the tags `fault`, `information`, `state change`, `configuration`, and `periodic notification` are available. Other tags can be added if necessary.

Depending on the use case or notification class, monitoring functions may need additional parameters. These parameters provide options for the execution of the monitoring function or define conditions under which managers have to be informed about events. Some monitoring functions do not need parameters (such as functions that inform about a reboot of a system), other functions may need several parameters.

The design is divided into three modules (see Figure 1). The configuration module offers a Web service interface, with which managers can administrate their subscriptions. The monitoring module performs the supervision of the resource. The notification module informs the manager on behalf of the monitoring module in case an event occurs.

To allow several managers to connect to the agent independently, managers use a Global Unique Identifier (GUID). In particular, this allows several managers to use the same monitoring functions. To identify a subscription, every instance of a pair of manager and monitoring function uses a unique handle, because a manager can subscribe to the same monitoring function more than once. The configuration interface offers the following functions:

- `subscribe( guid, monitorfunction, parameterlist )`
  Start a subscription to a monitoring function. If the subscription is successfully established, the manager will receive a handle that identifies the subscription.
- `unsubscribe( guid, handle )`
  Remove the subscription identified by the given handle.

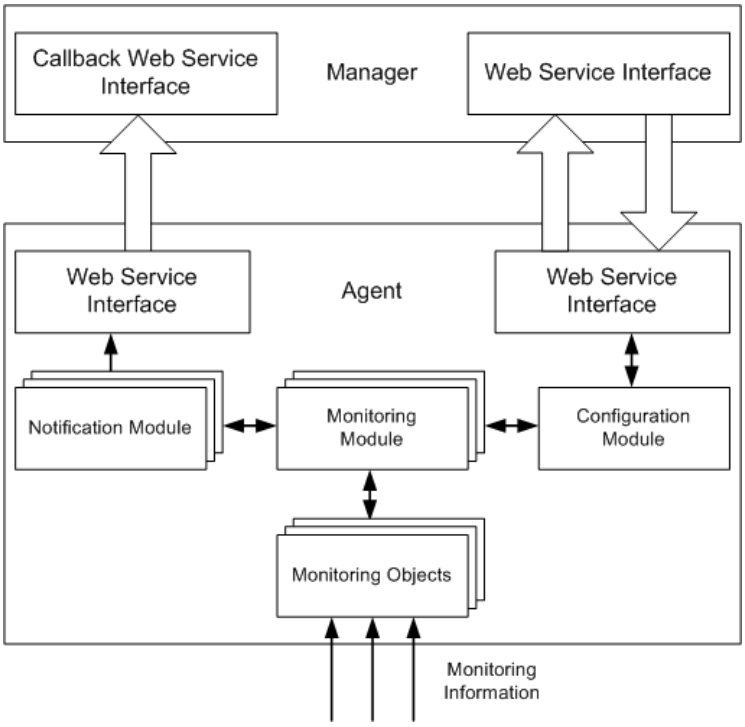

Fig. 1. Design of the monitoring system

- `reconfigure( guid, handle, parameterlist )`
  Change a subscription identified by the given handle and apply the given parameters.
- `getMonitorObjects()`
  Show all monitoring functions the agent provides.
- `getParameterInformation( monitoringfunction )`
  Retrieve information about the required parameters of the given monitoring function.
- `getRegisteredMonitoringInstances( guid )`
  Get a list of active subscriptions for the given manager.

To ensure flexibility and extensibility, all monitoring functions should be implemented in independent software modules, which communicate with the monitoring module using a well defined interface. These modules are called monitoring objects. Monitoring objects should be grouped and categorized with other similar monitoring objects in a tree-like fashion for clarity reasons, since there can be a large number of monitoring objects (see example shown in Figure 2).

Every manager must provide a call-back Web service interface. The notification module of the agent uses this interface (`pushNotification( guid, handle, messageClass, data )`) to send notifications to the manager.

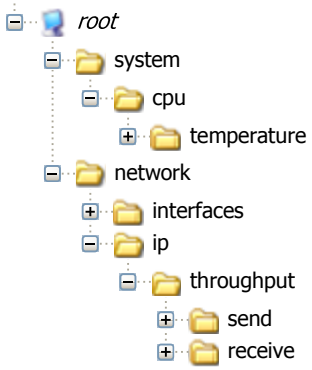

Fig. 2. Hierarchical organization of monitoring objects

4.3 Implementation

We implemented both a manager application and a prototype agent application in Java 5.0. For Web service support, we have used Webmethod's Glue Standard Edition [15]. The agent simulates a complete resource. Thus, besides the basic agent functions, the agent also contains a data generator to produce some realistic data for testing purposes.

Each monitoring object in the agent implements the method `monitor_run()`, which is called periodically by a monitoring thread. If this behavior is not desired for a specific monitoring object, the generic monitoring thread can be overridden by a more suitable one. The agent creates such a monitoring thread for each manager that has subscribed to a management function of the agent, as well as a notification thread. Having these two threads per manager makes the system more scalable and fault tolerant, because problems in one thread do not lead to problems in monitoring threads belonging to other managers. If a manager cancels its subscriptions at an agent, the corresponding threads will be terminated.

Monitoring objects are implemented in the package `agent.monitorObjects`. Within this package, the hierarchy presented in Section 4.2 can be implemented (e.g. `agent.monitorObjects.system.cpu.Temperature`), using arbitrary levels of nesting. Own objects can be added here. The agent application uses reflection to find all available management objects.

The manager application contains a graphical user interface (GUI) with which the administrator can connect to connect to the agents and delegate monitoring tasks to them. Furthermore, active subscriptions can be modified or cancelled. In case of an arriving event notification, it will display the notification in the manager window.

4.4 Evaluation

The correctness of the implemented prototype has been validated using a Debugger, JUnit [16] tests and experiments with several scenarios with multiple managers and agents.

1. 1 Manager ⇔ 1 Agent: One manager controls monitoring tasks for and receives notifications from a single agent. This test includes the call of all configuration functions on different monitoring functions with different parameter configurations. Moreover, sending and receiving of notifications is tested.
2. n Managers ⇔ 1 Agent: Multiple managers control monitoring tasks for a single agent. Besides the aspects tested in the 1-1 case, the n-1 test includes checks for data isolation, i.e. that each manager can only see and modify its own tasks.
3. 1 Manager ⇔ n Agents: A single manager controls monitoring tasks on different agents. This test is a generalization of 1-1 test.
4. n Managers ⇔ n Agents: Multiple managers control monitoring tasks for and receive notifications from multiple agents. This is the most realistic test case that combines the test cases 1-n and n-1.

Our initial tests have shown a good performance. However, as a next step, our approach must be tested in a large scale realistic testing environment in order to do a detailed performance analysis and comparison with our notification approaches.

With our approach, a manager can delegate large parts of the monitoring to the agent (management by delegation). This approach scales much better than a simple monitoring by polling. The impact on the network is lower, since the number of management messages in the network can be reduced significantly. Moreover, management applications only need to take actions if a certain event occurs (management by exception). This facilitates the development of management applications. Another advantage is the spreading of Web services. They are available for a large number of platforms and well known to a large developer community. Communication is based on open protocols (SOAP, HTTP, etc.).

On the other hand, our approach has certain drawbacks. Although efficient implementations exist, Web services still have quite a large footprint on machines due to the protocol overhead. The management by delegation also leads to a larger footprint on the agent, because the agent has to perform monitoring in addition to its actual tasks. An issue with the implementation is the vendor dependence. Glue standard Edition (in the form that we have used) is no longer available. However, the prototype can (and will be) ported to use an application server and SOAP engine such as Apache Tomcat and Apache Axis.

In Sections 2 and 3, we have presented several existing notification environments. Here, we briefly compare our approach with the other mentioned approaches (see Table 1).

Our approach is not linked to any Web services or management framework. Whether this is an advantage or a disadvantage depends on the use case. It will

	SNMP Traps	Syslog	WS-Eventing	WS-Base Notification	NETCONF	Our App.
Scope	network devices	system applications	generic	generic	network devices	generic
Framework	SNMP	none	WS-*	WSDM	NETCONF	none
Data Format	ASN.1 (BER)	syslog	XML	XML	XML	XML
Transport Protocols	UDP	UDP, others	SOAP (HTTP, HTTPS)	SOAP	SSH, BEEP, SOAP	SOAP
Web Services Support	no	no	yes	yes	optional	yes
Intermediaries	no	yes	no	yes	no	no
Subscription	no	no	yes	yes	yes	yes
Automatic Subscription Termination	n/a	n/a	timeouts	no	no	no
Transport Modes	push	push	push, others	push, pull	push	push
Event Classes	7	24	user	user	user	5+user

Table 1. Comparison between notification approaches

be more easy to use stand-alone but more difficult to integrate into an existing framework. Our approach is subscription-based and uses XML as the data format (like the other Web services approaches). It is not limited to a specific scope and easily extensible. For example, event classes can be added as needed. Note that in our approach event classes may overlap since we use tagging.

5 Conclusions

In this paper, we have discussed several asynchronous event notification approaches for network management. We have compared traditional approaches (SNMP Traps and syslog) to newer XML-based approaches that use Web services (except NETCONF, which uses SSH and offers only optional SOAP transport), including our own prototype implementation. Notifications should only be sent to managers that have subscribed to a certain event type. Furthermore, XML-based data encoding eases processing of received notifications, the use of Web services makes the development of management applications easier. Especially in autonomic environment following the Service Oriented Architecture (SOA), the availability of management functions as Web services is a big advantage. Therefore, the traditional approaches are not suited to such systems. All presented XML-based approaches follow the same basic principles (subscription, XML-encoding). However, there are differences in the details and in the use cases. Customers can choose the framework that best fits their requirements. However,

this freedom of choice has a drawback: interoperability. Different standards lead sooner or later to interoperability problems and away from an integrated management. This is the opposite of what the approaches promise.

Our future work will concentrate on analyzing possible interoperability problems and search integrative solutions. Furthermore, we will integrate Web service based notification systems into our autonomic management architecture [17].

References

1. Harrington D, Presuhn R, Wijnen B: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411, Enterasys Networks, BMC Software, Lucent Technologies, December 2002.
2. Bullard V, Murray B, Wilson K: An Introduction to WSDM. OASIS Commitee Draft wsdm-1.0-intro-primer-cd-01, AmberPoint Inc., Hewlett-Packard and Computer Associates International, February 2006.
3. McCollum R, et al.: Web Services for Management. WS-Management June 2005, DMTF, June 2005.
4. Enns R: NETCONF Configuration Protocol. RFC 4741, Juniper, December 2006.
5. Lonvick C: The BSD syslog Protocol. RFC 3164, Cisco Systems, August 2001.
6. McCloghrie K, Perkins D, Schönwälder J, et al.: Structure of Management Information Version 2 (SMIv2). RFC 2578, Cisco Systems, SNMPinfo, TU Braunschweig and others, April 1999.
7. Schönwälder J: Characterization of SNMP MIB Modules. Proc. 9th IFIP/IEEE International Symposium on Integrated Network Management (IM). Nice, France, May 2005.
8. Gerhards R: The syslog Protocol. Internet Draft <draft-ietf-syslog-protocol-19.txt>, Adiscon GmbH, November 2006.
9. Newman C, Klyne G: Date and Time on the Internet: Timestamps. RFC 3339, Clearswift Corporation and Sun Microsystems, July 2002.
10. Box D, Cabrera LF, Critchley C, et al.: Web Services Eventing (WS-Eventing). W3C Member Submission SUBM-WS-Eventing-20060315, Microsoft, IBM, TIBCO Software, BEA Systems and Computer Associates, March 2006.
11. Graham S, Murray B: Web Services Base Notification 1.2 (WS-BaseNotification). OASIS Working Draft wsn-WS-BaseNotification-1.2-draft-03, IBM and Hewlett-Packard, June 2004.
12. Chisholm S, Trevino H: NETCONF Event Notifications. Internet Draft <draft-ietf-netconf-notifications-04.txt>, Nortel and Cisco, October 2006.
13. Chisholm S, Curran K, Trevino H: NETCONF Event Notifications. Internet Draft <draft-ietf-netconf-notifications-02.txt>, Nortel and Cisco, June 2006.
14. Klie T, Wolf L: Autonomic Policy-based Management using Web Services. Proc. 2nd CoNext Conference. Lisbon, Portugal, December 2006.
15. Webmethods: Web Service Development. Product Web Site, 2006. URL http://www.webmethods.com/meta/default/folder/0000006047.
16. Meade E, Martin RC, Kohnke J: JUnit, Testing Ressources for Extreme Programming. Project Web Page, Object Mentor, 2004. URL http://www.junit.org.
17. Gu X, Klie T, Wolf L: A Proactive Policy-based Management Approach Towards Autonomic Communications. Proc. 4th IEEE Consumer Communications & Networking Conference (CCNC). Las Vegas, USA, January 2007. To appear.