RC 21770 (97933) (05/05/2000)
Computer Science/Mathematics

IBM Research Report

Dynamic Dependencies in Application Service Management

Alexander Keller, Gautam Kar

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

This page intentionally left blank.

Dynamic Dependencies in Application Service Management

Alexander Keller, Gautam Kar
IBM Research Division, T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, USA
E-Mail: {alexk|gkar}@us.ibm.com

Abstract

This paper addresses the role of dependency analysis in distributed management. It introduces the concept of dependency lifetime that traces the flow of dependency information from the design to installation to runtime stages of a service. Two categories of models, functional and structural, are used to represent this information as it flows from the design to the runtime stages. The paper describes how the dependency information at each of these stages can be retrieved and evaluated by management applications and presents an architecture to integrate these with existing network management platforms.

Keywords: Dependency Analysis, Application Service Management, Root cause Analysis, Impact Analysis, Problem Determination

1 Introduction

The identification of dependencies becomes increasingly important in today's networked environments because applications and services rely on a variety of supporting services that might be outsourced to a service provider. Failures occurring in lower service layers affect the quality of service of end-to-end services that are offered to customers. However, service dependencies are not made explicit in today's systems, thus making the task of problem determination particularly difficult. Solving this problem requires the determination and computation of dependencies between services and applications. One, therefore, has to deal with questions such as: what are the important characteristics of dependencies? In other words, when a managed entity X, such as a service or resource, depends on another managed entity Y (X is then termed **dependent** and Y **antecedent**), what are the properties of such a dependency that need to be recorded? How can we classify dependencies such that they can be used more efficiently to do root cause or impact analysis in fault management?

The notion of dependencies can be applied at various levels of granularity: For example, threads within a running application may be dependent on each other's operational output; a stored procedure within a database management system may be dependent on a lock administrator, etc. This paper does not consider such situations because there is a big difference between application *management* (focusing on application behavior observable from "outside") and application *debugging* (focusing on the internal behavior of an application). We consider only dependencies of the former type, i.e., that exist between different managed objects or components and, hence, are visible from outside an application.

The paper is structured as follows: Section 2 analyses the requirements on application dependency models by focusing on two typical service provider scenarios. It also gives an overview on related work in this area and identifies the deficiencies of existing standards and specifications. Section 3 classifies dependencies according to criteria that we have identified during our research. The methodology for determining and computing dependencies and our resulting architecture are presented in sections 4 and 5. Section 6 concludes the paper and presents issues for further research.

2 Requirements Analysis

Our first scenario deals with **managing outsourced services**, typically offered by *Internet* or *Application Service Providers (ASP/ISP)*. As depicted in figure 1, outsourcing services implies layered service provider hierarchies. At every layer, a service is accessed through a *Service Access Point (SAP)*. A SAP delimits the boundary between the different organizational domains and is the place where *Service Level Agreements (SLAs)* [5] are defined and observed. Usually, this is done at every layer by monitoring a set of specific parameters that are exposed by the provider. Dependencies between the different services are made explicit at the domain boundaries (in terms of SLAs) and can therefore be regarded as inter-domain dependencies.

The second scenario deals with the regular maintenance tasks that cannot be done "on the fly" and therefore

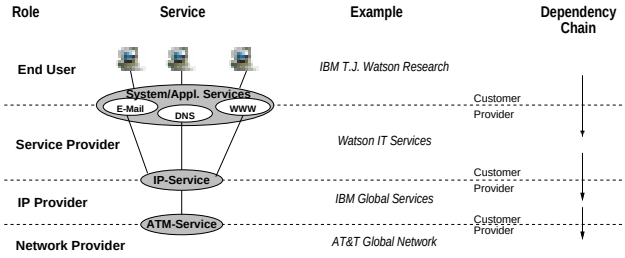

Figure 1: Dependencies in outsourced Services

affect services and their customers: Email servers get updated with a new release of their operating system, network devices are exchanged or upgraded with a new firmware version etc. In all cases, it is crucial for the network and server administrators to determine *in advance* how many and, more specifically, which services and users are affected by the maintenance. This is also known as **Availability Management**. This scenario allows us to identify another type of dependencies, namely dependencies that occur between different systems ("inter-system").

Both scenarios allow us to derive the following requirements and characteristics of dependency information:

1. Dependencies between *different* services are layered; furthermore, their dependency graph is directed and acyclic: The latter statement also reflects the authors' experience with IP-based networked services (such as DNS, NFS, DFS, NIS etc.) but there might be cases where mutual dependencies might occur in some systems: A "pathological" example for such a mutual dependency is a DNS server that mounts the filesystem in which its DNS configuration is stored via NFS from a remote system. It is the authors' belief that while such a configuration is technically possible, it reflects flaws in the system design because this leads to an unstable system whose bootstrapping might be non-deterministic and thus should be avoided. A dependency-checking application that discovers cyclic dependencies should issue a warning to an administrator.
2. Every dependency is visible at a customer/provider domain boundary and made explicit by means of SLAs; it follows that the number of observable dependencies is finite.
3. Dependency models must allow a top-down traversal of dependency chains.
4. Dependencies between different systems ("inter-system") are perceived as dependencies between the client and server parts of the *same service*. It is not possible that a client for service A issues requests to a server which provides a different service B.
5. suitable dependency models must (in addition to the top-down navigation allow a bottom-up traversal of dependency chains.
6. the number of dependencies between many involved systems can be computed but may become very large. From an engineering viewpoint, it is often undesirable - and sometimes impossible - to store a complete, *instantiated* dependency model at a single place. Traditional mechanisms used in network management platforms such as keeping an instantiated network map in the platform database therefore cannot be applied to dependencies due to the number and the dynamics of the involved dependencies. These two facts make it prohibitive to follow a "network-management-like" approach for the deployment of application, service and middleware dependency models. Instead, we propose to distribute the storage and computation of dependencies across the systems involved in the management process.

As an engineering response to the last item, section 5 presents an approach to tackle this problem. It is is built on the definition of two different kinds of dependency models:

A **Functional Model** that defines generic service (database service, name service, web application service etc.) dependencies and establishes the principal constraints to which the second model is bound.

A **Structural Model** containing the detailed descriptions of software components that realize the services (DB2 UDB 5.2, BIND 6.5, WebSphere Advanced Edition 3.0 etc.). It provides details w.r.t. the installed software and extends the amount of information provided by the Functional Model.

2.1 Related Work

The OpenGroup *Distributed Software Administration (XDSA)* specification [6] addresses the mechanisms for software distribution/deployment/installation by defining several commands (swinstall, swlist, swmodify etc.) and a software definition file format with many attributes. The latter identifies three kinds of relationships: prerequisite, exrequisite, corequisite. It should be noted that XDSA does not deal with *instantiated* applications and services and therefore does not represent means of determining the dependencies between components at runtime.

In the *Common Information Model (CIM)* [2], dependencies (being ususally perceived as a specific kind of association) are modeled as classes, thus allowing inheritance. The following dependencies are specified in the different CIM schemas: The *Core Schema* defines dependency types in terms of abstract classes that

deal with dependencies between SAPs, services, products and provide a generic means for associating context with them. The `Type of Dependency` attribute describes for a given service that its antecedent "must have completed", "must be started" or "must not be started" in order for a service to function. This is related to the notion of prerequisites, corequisites and exrequisites in XDSA. The *System Schema* refines the root class `CIM_Dependency` in order to deal with job destinations, host systems and file systems. The *Application Schema* defines two dependency types that describe an association between a service and an SAP and their implementations, respectively. Finally, the *Distributed Application Performance Schema* relates the definition, the metrics and the logical element that is instantiated to a "unit of work". All schemas have in common that their dependencies are derived from a single base class `CIM_Dependency` being defined in the Core Schema; the inheritance hierarchy is therefore flat.

It is fair to say that while each specification addresses the dependency problem in general, none of these specifications allows the determination of dependencies *at runtime*: In XDSA and CIM, the dependencies are well specified for the installation phase of a software product. However, these models provide no support as soon as an application gets instantiated, i.e., moves from the "installed" state to the "running" state. The main reason for the absence of satisfactory application management solutions is that comprehensive application management demands a large amount of management information, thus posing an additional development effort on the application developers [4, 3]. A good example for this is the *Application Management Specification (AMS)* [1] that provides an open standard for defining the management information needed for distributed applications. While AMS identifies a set of management information common to different kinds of applications and the means of specifying it using application description files, the application developer needs to provide the appropriate instrumentation.

3 Classification of Dependencies

Since dependencies come in different flavors and have varied characteristics, dealing with them in a systematic way can be facilitated by classifying them into groups with similar properties. Our approach consists in defining a coordinate system based on the following key characteristics of a dependency.

Space/Locality/Domain – how "far" is the antecedent from the dependent (i.e., sharing memory space, sharing the same node, sharing the same subnet, being located within the same domain etc.). This can also be referred to as "inter-domain" vs. "intra-domain" or "intra-system" vs. "inter-system" dependency.

Component Type – what the antecedent component actually is, i.e., a piece of hardware, an end system, a software package, a service, or a logical entity (a file system, a queue, a session). This distinction is important because different types of components tend to behave differently and to fail differently. One could argue that **Component Type** and **Activity** (see below) do not relate immediately to the computation of dependencies since they deal only with the component itself. However, our experience has shown that such information has to be recorded in order to facilitate problem determination and troubleshooting. It is therefore important to take the Type and Activity of a component into account, since software and hardware fail in different ways and require different approaches to correct the problem. This information is also valuable for configuration management.

Component Activity – whether the antecedent is "active" (such as a piece of hardware or software) and can be directly/explicitly queried, or "passive" (such as a file) which *by itself* cannot be queried or instrumented. Since only active components are likely to offer management instrumentation, this distinction is important for performance and fault management because an active component can be instrumented and thus queried directly. On the other hand, a passive component itself cannot be instrumented and must always have an "intermediary" acting on its behalf.

Dependency Strength – how strongly the dependent component depends on the antecedent resource. While it may seem not very useful to consider dependencies other than mandatory, it turns out that strength is a good metric to deal with intermittent (temporary) dependencies.

Dependency Formalization – what degree of formalization this dependency has and, thus, to which degree it can be determined automatically. For example, a dependency may be directly available from system information repositories such as ODM (AIX) or RPM (Linux) in machine-readable format (high degree of formalization); or a dependency may exist only in the notebook of a system administrator (very low degree of formalization). This dimension is important because it serves as a metric that helps to evaluate how expensive and/or difficult it is to acquire, identify, represent and track this dependency during the lifetime of the component. Another example for formal dependencies is found in the well-known configuration files of applications or networked services: The client part of the Domain Name System ("Resolver") is configured in the file */etc/resolv.conf*; this file contains not only the domain name of the client system but also information regarding the DNS servers and domains that are being

queried (and their order). It is therefore possible to compute the runtime dependencies of the Name Service. Detailed configuration parameters of a WWW server are found in the *httpd.conf*, *access.conf* and *srm.conf* files and allow to forecast the runtime behavior of a web server.

Dependency Criticality – how should this dependency be satisfied, in terms of availability of the resource/service this component depends on. *Dependency Criticality* can take the following values: *prerequisite*, *corequisite*, or *exrequisite*. It is important to note that, as the dependency moves along the *time* axis, the meaning and the semantics of these values change:

- *prerequisites* – components and services needed **before** this component can be used:
  - *installation time*: the components or resources must be installed before this component can be installed;
  - *runtime*: the services must run **and complete** (without encountering an error) before this component/service can start. A typical example for this is the need to obtain a valid software license from a license server before a word processor can be started.
- *corequisites* – components and services needed **in parallel** with this component:
  - *installation time*: a service must be installed/configured in conjunction with another service.
  - *runtime*: the service must be running when this component/service runs.
- *exrequisites* – components and services that **must not** be present:
  - *installation time*: in order for this component to be installed, services mentioned as *exrequisites* must not be installed;
  - *runtime*: in order for this service to run, those listed as *exrequisites* must not run.

Apart from these characteristics, it is important to consider two additional aspects that determine the behavior of dependencies but do not fit into the multidimensional representation:

Time: Dependencies may and usually do change from one point in the component's lifetime to another. Some may disappear – such as the need for disk space to install the component is gone once the installation is complete. Some may change – for example, the need for temporary disk space may not be gone but eventually decrease from the installation to the running phase of an application. And some dependencies may come up

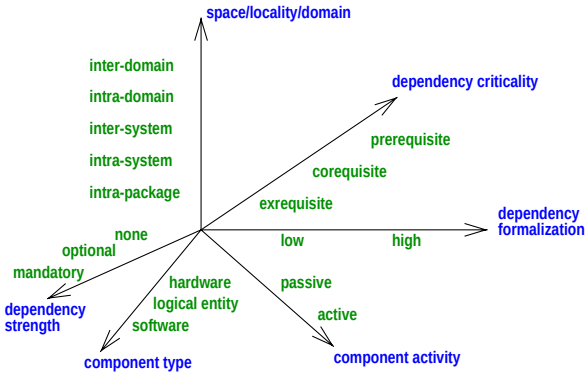

Figure 2: Multidimensional Space of Dependencies

that were not there before – such as the need for remote components which matters when the application runs, but should not and does not prevent the installation from succeeding. For example, IBM WebSphere may depend on the availability of a certain amount of disk space at installation time — however this dependency may disappear as the application lifecycle moves beyond installation. Another example – dependency upon DNS – appears only after the installation is complete and possibly past the configuration stage.

Dependency Lifecycle: It is useful to distinguish between *functional* and *structural* dependencies. Usually, as a dependency moves along the *time* axis, it is *instantiated* further, accumulating details and evolving from functional to structural. A *functional* dependency is an association between two entities, typically captured first at design time, which says that one component requires some services from another. A *structural* dependency contains more detailed description and is typically captured first at deployment or installation time.

One also has to differentiate between dependencies crossing the local host boundary, and those that relate to components within one host. Thus, another space-related classification of dependencies deals with the respective layers of the components related by the dependency in question. The dependent component may be on the same layer (usually *inter-system*) – for example, a database client application depends on a database server. If the antecedent component is located at a lower layer, we speak of *intra-system* dependencies, e.g., a Web browser depends on the TCP service.

It is important to know how a given dependency can be obtained for the following reasons:

- to evaluate the cost of obtaining it vs. the benefit it offers;
- to evaluate the accuracy of the information obtained.

Dependency information can be provided with the component, or it can be computed. The best source of such information is the component developer, as he knows firsthand what services his product requires, or can offer. Examples of how dependencies can be obtained are:

- the application explicitly lists its dependencies;
- the information is acquired from the environment (e.g., system repository, configuration and installation files, etc.).

Dependencies can be provided by the application developer and be either delivered as part of the documentation, or included in a component package file (in a machine-readable format, suitable for immediate acquisition and use by management applications). Tivoli AMS is an example of such a system. The cost of obtaining such dependencies is relatively low, and their accuracy is high. However, this is today a rather unusual case as the majority of applications is not "management-enabled". Thus, our approach is based on the second alternative: The sources of our dependency information are system-wide repositories of software package dependencies, maintained by the operating system. This information is complemented by evaluating the content of configuration files and by observing the signals raised by the operating system. The additional advantage is that the dependencies of a particular software w.r.t. the environment can be determined.

4 Dependency Analysis

The analysis in the previous section has pointed out that a major requirement for the automated management of distributed application services is to have a record of their dependencies on lower layer services and resources.

Considering the fact that a majority of application services run on UNIX and Windows NT-based systems, it is worth analyzing the degree to which information regarding applications and services is already contained in the operating systems. The underlying idea is as follows: if it is possible to obtain a reasonable amount of information from these sources, the need for application-specific instrumentation can be greatly reduced. Our approach recognizes the fact that system administrators successfully deploy applications and services without having access to detailed, application-specific management instrumentation.

WindowsNT/95/98 systems and UNIX implementations such as IBM AIX and Linux have built-in repositories that keep track of the installed software packages, filesets and their versions. AIX *Object Data Manager (ODM)*, Windows *Registry*, and Linux *Red Hat Package Manager (RPM)* are examples for these system-wide configuration repositories. In this paper, we will concentrate on ODM. However, we have verified the applicability of our approach to the other repositories as well.

Usually, repositories serve as the basis for software installation tools such as *InstallShield* (for Windows systems) or the AIX *Systems Management Interface Tool (SMIT)*. Moreover, they can be regarded as knowledge bases that contain important information with respect to the compatibility of services and applications. The fact that a specific software package must already be present on a system so that another package can be installed successfully, implies that the service implemented by the latter package depends on the service implemented by the former. In other words, if a specific software package has other software packages listed as installation prerequisites, we can infer that this dependency relationship is also valid for their instantiated counterparts, i.e., services and applications.

A simple example will serve to illustrate this: if the system repository indicates that a specific Web server has a distinct TCP/IP implementation as a prerequisite, this essentially means that:

1. A **functional**, i.e., generic and implementation-independent relationship model for the service categories "WWW" and "TCP/IP" can be established. The model describes services in terms of the functionality they provide and on which other services they depend. The fact that the WWW service depends (among others) on the availability of the TCP/IP service implies that a functional dependency relationship between the services "WWW" and "TCP/IP" is defined (by means of the description in the prerequisites list) and enforced by the installation routine.
2. **Structural**, i.e., specific and implementation-dependent management information is available. An appropriate algorithm for automated problem determination for a malfunctioning WWW service would have to check whether the operational states of a specific web server and of its underlying TCP/IP stack are "up". This is possible because the dependency relationships of the majority of networked services are explicitly listed in the system repository.

Discovering and enumerating the dependency relationships that applications have on lower layer services in a networked environment is a difficult problem. It has both a static and dynamic aspect, that is, dependencies identified at application install time and those discovered at runtime. The functional dependency model can be used to describe static dependencies between application and service categories. The structural part cap-

tures dynamic information related to concrete service implementations.

We will now take a look at how configuration information for software packages is represented in ODM. The following entry for the successfully (`state = 5`) installed TCP/IP client component[1] (`bos.net.tcp.server`) – being part of the operation system networking software (`bos.net`) of IBM AIX – indicates that this package (version `4.2.1.0`) replaces and renames the previously installed package `bosnet.tcpip.obj` version `4.1.0.0`.

```
lpp_name = "bos.net.tcp.server"
update = 0
name = "bos.net"
state = 5
ver = 4
rel = 2
mod = 1
fix = no
ptf = ""
sceded_by = ""
prereq = "*prereq bos.rte 4.2.0.0,
          *prereq bos.net.tcp.client 4.1.0.0"
description = "TCP/IP Server"
supersedes = "bosnet.tcpip.obj 4.1.0.0"
```

Additional information for applied fixes/patches and their descriptions is also found in this template. One particularly important part of this data structure is the entry "`prereq`" (prerequisites) because it mandates which other software packages must already be present in the system so that this component can be successfully installed. In the following sections, we will describe how we make use of this information for our purposes. Since every software package installed on AIX is required to list its properties in this machine-readable format, we can therefore assume that the dependencies cover a comprehensive set of software packages.

Our analysis has shown that system repositories such as ODM represent a rich source of application service management information, not only regarding the configuration of installed applications but also for determining dependency relationships between applications and services. Note that this large amount of information can be obtained *without any* specific instrumentation of the components. The only requirement is that the application components be described using a **service description template** that has been developed by us and whose content can be provided to a large degree by today's system information repositories.

5 Dependency Architecture

In this section we will briefly illustrate the role of dependencies in the management of services in a typical

[1]The term "LPP" in the template stands for "licensed program product" and denotes a service component.

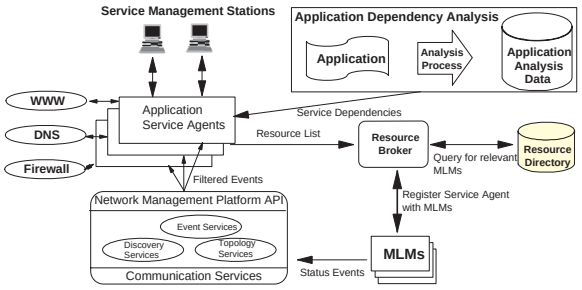

Figure 3: Components of the Architecture

ISP environment. Our overall architecture is depicted in figure 3: Off-the-shelf network management platforms and Mid-level Managers (MLM) provide the basis of this architecture and offer services such as event reception and forwarding, resource discovery functions or topology services. We assume that through offline analysis – based on the methodolgy and approach presented in section 4 – a database of static dependencies in constructed. This collected data describes, for each end-to-end application service, the dependecies it has on lower level application and network layer services and components.

The **application service agent** is the focal point for managing an individual service offering. From an implementation viewpoint, it is like an application operating on top of a network management platform. If, for instance, a service provider has three different service offerings (DNS, web-based content hosting and shared firewall) our design yields three service management agents, each responsible for one offering. The agent receives event notifications from the MLMs through the platform API and updates the view of the service that it maintains. This view can best be represented by a multi-level resource tree, where the elements in one level are dependent on the availability and status of elements at the next lower level. One way to use the service view is to represent it graphically on one of the service management stations where a service manager can observe the status of the service and do typical drill down operations for troubleshooting.

The functional dependencies yield a generic service model while the structural part provides detailed information on the involved components. While the functional model is stored at the MLMs and the management platform, the service management agent maintains a structural view for every individual customer. This leads to a high degree of distribution and is the main reason for the scalability of our approach; thus, an outage in any resource can be rapidly correlated with complaints received from that customer. During initialization, a service agent obtains the configuration file

generated at the application dependency analysis phase, which is used to establish the functional model.

The **Resource Broker** serves as the main access to the Resource Directory. The two entities which use the Resource Broker are the Application Service Agent and the Mid-level Manager. As part of its initialization procedure, the Application Service Agent sends its list of resources contained within its service view. The Resource Broker searches the Resource Directory and associates with this list a set of MLMs that are responsible for monitoring the resources. It then communicates with the relevant MLMs in order to register the service agents with the MLMs. The MLMs use this registration lists to direct events pertaining to the resources under their monitoring control.

The main function of the **Resource Directory** is to maintain a list of all resources that are monitored by a particular MLM. It responds to queries from the resource broker and provides persistent storage of the records.

We will now describe the information flow that takes place in order to produce status views of application service offerings.

An Application Service Agent in charge of a particular application service sends a request to the Resource Broker in order to determine the status of the resources the service depends on. It provides the Resource Broker with a list containing the resource identifiers of all the resources this service requires. This dependency list is generated during the application dependency analysis phase.

The Resource Broker queries the directory using the above resource list and obtains a list of all MLMs that are responsible for monitoring the set of resources in this list. Upon obtaining the above list of MLMs, the Resource Broker contacts each of the MLMs by sending the Application Service Agent ID and the associated resource list, i.e., the set of resources for which the particular MLM is responsible. At this point, when an resource related event is generated, each MLM monitoring the resource, knows where, i.e., to which Application Service Agent the events should be forwarded.

For efficiency reasons, events from the MLMs are filtered by the event services of the management platform. Depending on the agent IDs attached to the events, the platform forwards them to the responsible Application Service Agent, which, in turn, uses the information in the event to update the status information in the service view.

One of the advantages of the approach presented above is that it allows for incremental, low cost, staged deployment. Most ISPs start by offering simple connectivity services and then evolve into application service providers. Typically, they start by deploying standard, platform-based, network operations and management systems. Using our approach, these systems can be augmented incrementally to use dependency information for the management of services, thus allowing ISPs to preserve and leverage their investments in the initially deployed operational systems.

6 Conclusion and Outlook

A key requirement for application service management, highlighted in this paper, is the identification, computation and representation of dependency information. Since dependencies come in different flavors and have varied characteristics, dealing with them in a systematic way can be facilitated by classifying them into groups with similar properties. We have developed such a classification which helps to identify the various aspects of dependencies. The approach for identifying and computing dependencies presented in this paper is pragmatic and based on a static dependency analysis that yields information on entities within a system (Intra-system) and between peer entities of a service (Inter-system). Finally, we have provided a description of the components of our Application Service Management Architecture and the information flows.

The identification of dependencies is a prerequisite for the deployment of troubleshooting services that capture fault management knowledge contained in fault documentation systems. The authors are currently engaged in further research in modeling and implementing management services for optimizing the traversal of dependency structures.

References

[1] Application Management Specification. Version 2.0, Tivoli Systems, November 1997.

[2] Common Information Model (CIM) Version 2.2. Specification, Distributed Management Task Force, June 1999.

[3] M. Katchabaw, S. Howard, H. Lutfiyya, and A. Marshall. Making Distributed Applications Manageable through Instrumentation. *The Journal of Systems and Software*, (45), 1999.

[4] A. Schade, P. Trommler, and M. Kaiserswerth. Object Instrumentation for Distributed Applications Management. In *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms (ICDP'96), Dresden, Germany*. IFIP, Chapman and Hall, 1996.

[5] D. Verma. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.

[6] Systems Management: Distributed Software Administration. CAE Specification C701, The Open Group, January 1998.