Proceedings of the *IEEE International Conference on Communications (ICC) 2001*,
Helsinki, Finland, June, 2001

Determining Service Dependencies in Distributed Systems

Alexander Keller, Gautam Kar
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
E-Mail: {alexk|gkar}@us.ibm.com

Abstract— **We describe an architecture and its implementation for retrieving and handling dependency information from various managed resources in a web-based environment. The core of our architecture is a dependency query facility that allows the application of queries and filters to dependency models; its output is a consolidated dependency graph that can then be used as input for event correlators and various management applications to display service topologies or to perform additional problem determination tasks. The definition of an XML-based notation for specifying dependencies facilitates the sharing of information between the various components involved in the process.**

Keywords— **Service Management, Application Management, Problem Determination, Dependencies, Fault Management**

I. Motivation

The identification of dependencies has become 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. In order to perform end to end fault management, which includes tracing the root cause of a problem manifested at a customer service offering, it is necessary to navigate through dependency information, which relates how services in one layer of a distributed system affect those in another layer. However, in practice, service dependencies are not made explicit in today's systems, thus making the task of problem determination difficult. Solving this problem requires the determination and computation of dependencies between services and applications. For our discussion, we call services that depend on other services **dependents**, while services on which other services depend are termed **antecedents**. It is important to note that a service often plays both roles (e.g., a name service is required by many applications and services but is dependent on the proper functioning of other services, such as operating system and network infrastructure), thus leading to a **dependency hierarchy** that can be modeled as a directed graph.

What is needed is a dynamic model reflecting the dependency relationships between services; in addition, a management system should be capable to provide various mechanisms to select parts of a dependency model according to different criteria.

We have designed and implemented a system that allows to determine and compute such dependencies. It provides a uniform interface to query service and dependency information across the systems of a distributed environment and can be used by various fault and topology management applications or event correlation systems.

The paper is structured as follows: Section 2 analyses the requirements on service dependency models by focusing on two typical service provider scenarios. It also gives an overview on related work in this area. The methodology for determining and computing dependencies and our resulting architecture are presented in sections 3; our proof-of-concept implementation is described in section 4. Section 5 concludes the paper and presents issues for further research.

II. Requirements Analysis

Our first scenario deals with **managing outsourced services**, typically offered by *Internet* or *Application Service Providers (ISP/ASP)*. Outsourcing services leads to layered service hierarchies where, e.g., the services of an ASP depend on the IP-connectivity offered by an ISP, which, in turn, relies on the wide-area network of a telecom carrier. 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)* [7], [9] are defined and observed. Usually, this is done at every layer by monitoring a set of specific parameters that are exposed by the provider. In case of an error or performance degradation in an upper-layer service, it is necessary to traverse the service hierarchy from the top to the bottom to identify the **Root Cause** of the problem.

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

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 configura-

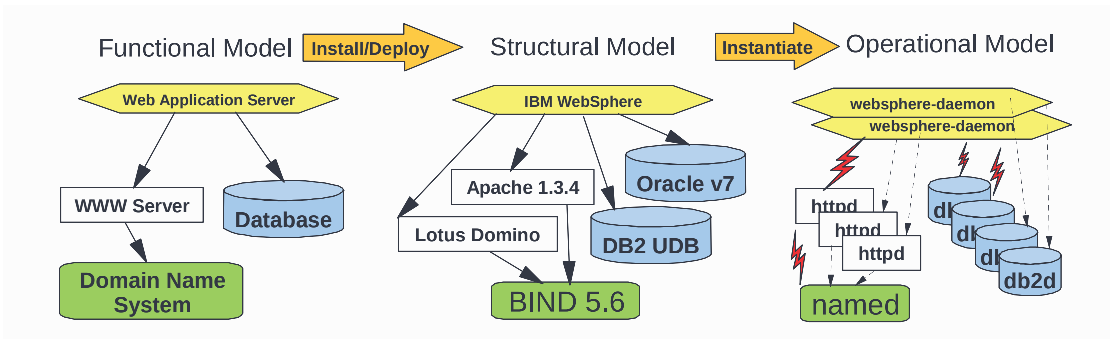

Fig. 1. Dependency models related to the service lifecycle

tion 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 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. Dependency models must allow (in addition to the top-down navigation) 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 III presents an approach to deal with the aforementioned characteristics and requirements. It is is built on the definition of three different kinds of dependency models, depicted in figure 1, that reflect the service lifecycle:

A **Functional Model** that defines generic service (database service, name service, web application service etc.) dependencies and establishes the principal constraints to which the models mentioned below are bound: While they add more detailed configuration and runtime information to an existing functional model, it is not possible for them to introduce new dependencies among service classes. This constraint is necessary to prevent the creation of loops in the graph. The functional model contains a generic service topology and is stored in the database of a management system; it can be modified by a system administrator.

A **Structural Model** containing the detailed descriptions of software components that realize the services (DB2 UDB 7.1, BIND 6.5, WebSphere Advanced Edition 3.5 etc.). It provides details w.r.t. the installed software and extends the amount of information provided by the functional model. The structural model reflects the actual software configuration present on every single system and can be retrieved from system configuration repositories such as the Windows Registry, the Linux RedHat Package Manager or the AIX Object Data Manager. This model is therefore kept at the managed resources.

An **Operational Model** that reflects the bindings between running service instances addresses the runtime stage in the services’ lifecycle. Since the amount of highly dynamic object instances and their dependencies is excessively large, it is not appropriate to keep the operational model in one place. Instead, we introduce a *Dependency Query Facility* (described in more detail in section III) that computes *the relevant part* of the operational dependency model on demand; i.e., it determines for a given service instance what its antecedents (or dependents) are.

A. Acquiring Dependencies

There are many different ways to gather service dependency information in a distributed environment; the most common approaches are listed below.

- The straightforward way is to provide appropriate instrumentation within the applications and services themselves; the problem is that none of today’s applications is able to provide this kind of information at an acceptable granularity.
- Another approach consists in instrumenting the communication protocol stack and/or some shared libraries of the host system to intercept the communication between different parties in order to infer potential dependencies. The resulting information could be either provided by a specific “dependency agent” or given out as flat files.

- In [5], an approach is described that makes use of the information stored in system configuration repositories for generating appropriate service dependency information.
- A technique used in system and protocol design, which has only recently been applied to service and application management is the active perturbation of components within a system (i.e., injecting faults in a controlled manner and observing the behavior of the components) while running synthetic transactions against it. [1] describes how active perturbation can be used to obtain the required dependency information; however, great care has to be taken if this technique is used on production systems.
- Other approaches come from the area of Artificial Intelligence. [3] uses Neural Networks to automatically derive dependency information by looking at system's behavior over time.
- Finally, a *CIM Object Manager (CIMOM)*, as proposed by the Distributed Management Task Force (DMTF) could be used to expose the necessary information. The CIM Core Model [2] provides an association class *CIM_Dependency*, from which several subclasses are derived.

Our work relies mainly on extracting information from system configuration repositories and combining it with the results obtained from perturbing some components of the distributed system under typical workloads. This gives us the advantage of gathering a lot of useful information for a wide range of applications and services (typically between 20 and 60 per system) without requiring them to be instrumented.

B. *Related Work*

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. A lot of research addressing such fine-grained dependencies *within* applications has been carried out in the area of software engineering; identifying such dependencies, however, requires the availability of the application source code. The service management environment addressed by this paper does not consider such situations because the prerequisites are different: The source code of applications implementing a service is usually not available to the administrator of a distributed system and it is not his duty to debug specific services but to keep the overall system running. There is a big difference between application *debugging* (dealing with the internal behavior of a service) and application *management* (focusing on service behavior observable from "outside"). We consider only dependencies of the latter type, i.e., dependencies between different managed objects and, hence, visible from outside an application that implements a service.

Previous work in the systems management area on identifying service dependency information has mainly been within the scope of event correlation (see e.g., [4] and [6]). However, the descriptions have always been in a proprietary format since their use was confined to a single management tool, namely an event correlator and thus could not be shared among different entities involved in the management process. Since it is unlikely that the different parties involved in fault management of outsourced applications use the same toolset for tracking dependencies, it is of fundamental importance to specify and implement a mechanism that determines dependencies and is able to expose this information in an open format. The following sections describe our approach to solve this problem.

III. Architecture

Our distributed three–tier architecture, depicted in figure 2 addresses the issue of dealing with potentially highly dynamic dependency relationships among a very large number of components. It follows a "divide and conquer" approach, which is usually the way of choice for dealing with scalability problems in distributed systems.

We assume that the managed resources (depicted in the right part of the figure) are able to provide XML descriptions of their system inventory and their various dependencies (for details on how this information can be acquired see section II-A).

In the center of the figure is the core component of our architecture: The **Dependency Query Facility**, triggered by queries of the management system using Java *Remote Method Invocation (RMI)*, processes them and sends the results back to the manager. Its main tasks are as follows:

- Interacting with the management system. The management system issues queries to the API of the Dependency Query Facility. The API exposes a flexible "drill–down" method that, upon receiving the identifier of a service, returns:
  - either descriptions of its *direct antecedents*, i.e., the first level below the node representing the service, or
  - the *whole subgraph* below the node, or
  - an *arbitrary subset* of the dependency graph (levels *m* to *n* below a given node).

A "drill–up" method with the same facilities, targeting the dependents of the service, is also present. In addition, methods for

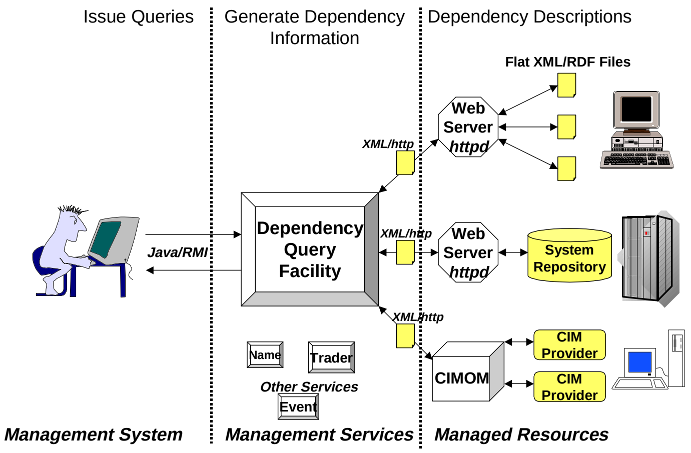

Fig. 2. Architecture of our Dependency System

gathering and filtering information for classes and properties of managed objects are available.

- Obtaining dependency information from the managed resources (by issuing queries over http) and applying filtering rules (as specified by the manager) to it.
- Combining the information into a data structure that is sent back to the manager as XML document.

The description of our implementation is given in section IV. It should be noted that due to its fully distributed nature, the architecture aims at keeping the load on every involved system as low as possible. It completely decouples the management system from the managed resources and encapsulates the time-consuming *filter* and *join* operations in the dependency query facility, which can be replicated on various systems. We are therefore able to achieve a maximum level of parallelism for query operations, since the selection of an instance of the facility can be done flexibly by the management system.

Another important advantage of our architecture is that the (very large and highly dynamic) operational dependency model is not stored in a specific place, but computed on demand and stepwise from the various structural models, located at the managed resources. The management system therefore always receives the most recent information (but is free to store the obtained information according to elaborate caching policies).

IV. Prototype Implementation

In order to demonstrate the concepts presented in this paper, we have put togther a prototype of a simplified three-tier e-commerce environment consisting of a web-based application server, where the business logic (a fictituous Internet storefront application) is implemented, and a back-end database. The environment, schematically depicted in figure 3, consists of a fairly heterogeneous mix of AIX and WindowsNT servers. As a first step, we have developed algorithms to access the repositories of the individual machines to extract relevant information and construct dependency graphs for services contained within one machine environment. These graphs are represented and stored as XML documents. Next, we have built a dependency query facility (see figure 2) that, upon request from a management application, obtains the current dependency information of the different nodes in the form of XML documents, and manipulates this information by means of XPath statements to build the operational dependency model as a basis for performing root cause and impact analysis.

Figure 3 illustrates that dependencies between different services are represented as objects (large black dots in the figure) themselves. This is necessary because the notion of dependencies is very coarse and needs to be refined in order to be useful. Examples for this are the *strength* of a dependency (indicating the likelihood that a component is affected if its antecedent fails), the *criticality* (how important this dependency is w.r.t. the goals and policies of an enterprise), the *degree of formalization* (i.e., how difficult it is to obtain the dependency) and many more. While it is out of the scope of this paper to establish a

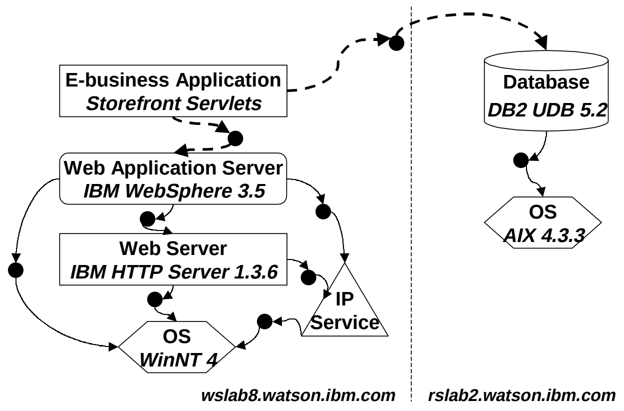

Fig. 3. Sample service dependency graph

taxonomy for dependencies, there is a need to add attributes to dependencies that allow their qualification, and accordingly, a need to reflect these attributes in the dependency representation.

Every element of the graph (the services and their attributes, the links between services and their dependencies, and the attributes of the dependencies) are represented as elements of an XML document. The core principles of our approach are as follows:

1. **Use the XML parsing capabilities for representing dependency information:**
When one (or many, stemming from various resources) XML document is read by a parser, a parse tree is generated where the managed objects, their attributes and dependencies become nodes in the parse tree.

2. **Use the Resource Description Framework (RDF) to represent directed acyclic graphs in XML:**
Following a hybrid RDF/XML approach and taking advantage of the RDF [8] capabilities to annotate nodes with links, we are able to map our dependency models into a representation that allows us to take advantage of the aforementioned XML capabilities. Section IV-A illustrates this by means of an example.

3. **Use the query facilities of the XML Path Language (XPath) to select and filter managed objects:**
XPath [10] provides a powerful query language that operates on XML parse trees (generated by DOM parsers) and allows to select and filter nodes of the parse tree according to their position and their properties. We have used XPath for performing sophisticated queries on dependency models (see section IV-B).

A. RDF Representation of Services and their Dependencies

We will now present by means of an example how our approach can be applied to represent services and their dependencies in XML/RDF. More precisely, we show the content of the document that specifically represents the dependency of `Storefront Servlets` on `IBM WebSphere 3.5` on the one side, and on `DB2 UDB 5.2` on the other. These dependencies are marked as dashed arrows in figure 3.

By definition, the header of every document starts with the

XML tag (line 1 of the following listing), followed by links to our dependency schema (line 2) as well as the RDF syntax and schema definitions (lines 3 and 4). The body of the document contains the service definition start and end tags (line 5, resp. 29), its attributes (lines 6 to 12) and two dependencies (lines 13 to 20, resp. 21 to 28). The document closes with the RDF end tag (line 30). Note that all pointers to descriptions of antecedents are URIs, thus making their location (local or remote) completely transparent to the dependency query facility.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rdf:RDF xmlns:ds="wslab4/DependencySchema#"
3   xmlns:rdf="www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:rdfs="www.w3.org/2000/01/rdf-schema#">
5    <ds:Service>
6    <ds:name>E-business Application</ds:name>
7    <ds:caption>Storefront Servlets</ds:caption>
8    <ds:identifier>catalogServlets</ds:identifier>
9    <ds:description>myCatalogApp</ds:description>
10   <ds:version>3</ds:version>
11   <ds:release>1</ds:release>
12   <ds:processName></ds:processName>
13   <ds:dependency>
14     <ds:ServiceDependency>
15       <ds:antecedent rdf:resource=
16         "http://rslab2/xmlrepos/db2.xml"/>
17       <ds:generated>automatic</ds:generated>
18       <ds:label>App_DB_Dependency</ds:label>
19     </ds:ServiceDependency>
20   </ds:dependency>
21   <ds:dependency>
22     <ds:ServiceDependency>
23       <ds:antecedent rdf:resource=
24         "http://wslab8/xmlrepos/websph35.xml"/>
25       <ds:generated>automatic</ds:generated>
26       <ds:label>App_AppServer_Dependency</ds:label>
27     </ds:ServiceDependency>
28   </ds:dependency>
29  </ds:Service>
30 </rdf:RDF>
```

B. *Querying Dependency Graphs with XPath*

Every XPath query describes a 'path' through the virtual tree structure of the XML document that is generated by a DOM parser. Each step on the path consists of:

- an axis: the 'search direction', e.g., towards the `child` or `ancestor` nodes,
- a node test: the name of the nodes (i.e., the tag–name) to be chosen, and
- one or more predicates that apply filters to the result. The predicate itself may consist of further XPath–expressions.

The simple XPath query /descendant::ds:Service[@rdf:about=*ID*] selects a certain element description from an XML document: The axis descendant specifies to search anywhere in the document below the current node (in this case the root node). After '::' follows the name of the desired node (ds:Service) and the filter predicate (in square brackets), which specifies to select only nodes with an attribute rdf:about that has a certain value (*ID*).

We have implemented more complex XPath statements to perform top-down and bottom-up traversals of dependency graphs, allowing the identification of both antecedents and dependents of a given service, respectively. This information can then be used by problem determination applications to pinpoint the root cause of an outage.

V. Conclusion and Outlook

We have presented a novel approach for managing service dependencies with XML, XPath and RDF. The need for applying these general–purpose technologies to the area of service and application management stems from the fact that, despite related work in the area of event correlation, no previous work has dealt with describing and exposing dependency information in a uniform way so that various management applications are able to use it. This is necessary in today's environments where outsourcing of services results in a vast amount of dependencies among services that are also highly dynamic.

We have combined several XML technologies and are therefore able to represent dependency graphs in a way that they can not only be parsed by common off the shelf XML parsers, but be also queried with the powerful XPath facility. This allows us to implement an efficient mechanism for querying a potentially very high number of managed objects in parallel for their attributes and dependencies. Our prototype implementation has shown that queries for (recursive) drill–up or drill–down operations are surprisingly compact and relatively easy to write. The problems we experienced during our work are mainly related to XML and, especially, RDF parsers, which are still in early stages of development.

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 on designing and implementing such management services.

References

[1] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Application Environment. In N. Anerousis and G. Pavlou, editors, *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE Publishing, May 2001.

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

[3] C. Ensel. Automated Generation of Dependency Models for Service Management. In *Workshop of the OpenView University Association (OVUA'99)*, Bologna, Italy, June 1999.

[4] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. In *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM '98)*, October 1998.

[5] G. Kar, A. Keller, and S.B. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. In J.W. Hong and R. Weihmayer, editors, *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS'2000)*, pages 61–75. IEEE Press, April 2000.

[6] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM 97)*, pages 583–596, May 1997.

[7] L. Lewis. *Service Level Management for Enterprise Networks*. Artech House, 1999.

[8] Resource Description Framework (RDF) Schema Specification 1.0. W3c candidate recommendation, W3 Consortium, March 2000.

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

[10] XML Path Language (XPath) Version 1.0. W3c recommendation, W3 Consortium, November 1999.