

## Classification and Computation of Dependencies for Distributed Management

Alexander Keller, Uri Blumenthal, Gautam Kar  
IBM T.J. Watson Research Center, Hawthorne, NY, USA  
E-Mail: {alexk|uri|gkar}@us.ibm.com

### Abstract

*This paper addresses the role of dependency analysis in distributed management. The identification of dependencies becomes increasingly important in today's networked environments because applications and services rely on a variety of supporting services which might be outsourced to a service provider. 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 the dependencies between services and applications. A key contribution of the paper is a methodology for making IP-based services and applications manageable that have not been designed to include management instrumentation (which is the case today for almost every application and service). Unlike other approaches, it is not necessary to modify the application code. Instead, our approach yields a technique that enumerates the characteristics and interdependencies of applications and services, thus permitting the derivation of appropriate management information.*

### 1 Application Service Dependencies

Application services are generally composed of a set of distributed components, each of which contributes a specific function to the sum total of the services provided by the application. Thus, a fault or malfunction occurring in any of the components can lead to a problem in the end-to-end service that the customer perceives. In addition, application components function in close cooperation with the elements that comprise the environment e.g., the communication network, the operating system, various middleware such as databases, messaging functions and their services, etc. Thus, determining the source of a problem involves finding the root cause which may lie in any of the components and services that contribute to the end-to-end service to the customer.

Dependencies represent consumer/provider relationships between various cooperating components in a distributed system. When one component requires a service performed by another component in order for it to execute its function, this relationship between the two components is called a **dependency**. For the purpose of our discussion, when a managed entity A (such as a service or an application component) depends on a managed entity B, we say that A is the **dependent** and B is the **antecedent**.

The notion of dependencies can be applied at various levels of granularity: Each thread within a running application

may be dependent on each other's operational output. This paper does not consider such situations because there is a distinct difference between application *management* (focusing on the application behavior observable from "outside") and application *debugging* (focusing on the internal behavior of an application). We consider only dependencies of the former type.

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 several criteria that we have identified. A repository-based methodology for determining and computing dependencies is presented in section 4. Section 5 concludes the paper and presents issues for further research.

### 2 Requirements Analysis

#### 2.1 Management of outsourced Services

The modularity of IP-based services (e.g., EMail, Name Service, World Wide Web) presents opportunities of outsourcing IT activities for cost savings. As in traditional networking, end-to-end services residing in the upper layers of a distributed system depend on those offered by lower layers. This leads to layered service provider hierarchies as depicted in the left part of Figure 1 where a service provider is at the same time the customer of another provider: The provisioning of end-user services such as Email, WWW and DNS therefore depends on the availability of the IP service which, in turn, depends on an ATM service. In general, at every layer, a service is accessed through a *Service Access Point (SAP)*; in this scenario, an SAP also delimits the boundaries between the different organizational domains and is the place where *Service Level Agreements (SLAs)* [11] 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 (ideally in terms of SLAs) and can therefore be regarded as **inter-domain dependencies**.

In this context, customers are most interested in

1. observing performance degradations and outages of subscribed services,
2. tracking down the root cause of the problem by traversing

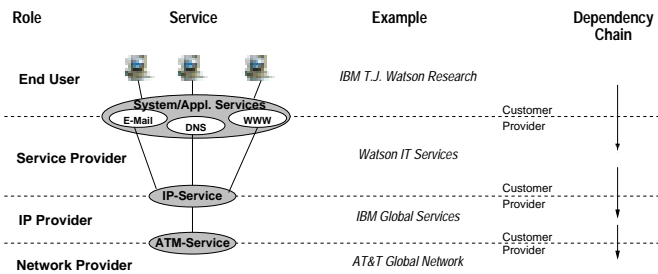


Figure 1: Dependencies between outsourced Services

the different layers of the dependency model from the top to the bottom. This (recursive) traversal crosses domain boundaries.

This leads to a top-down traversal of the dependency chain; an appropriate dependency model therefore must allow a “drill-down”.

## 2.2 Availability Management

Our 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 basically a rephrasing of the availability problem. The assumption that services are layered is fundamental for analyzing such impact in order to schedule appropriate maintenance windows and to notify the affected users in advance. Performing such an impact analysis requires the traversal of the dependency model from the bottom to the top.

This scenario allows us to identify another type of dependencies, namely dependencies that occur between different systems (**inter-system**): The maintenance of an Email server obviously affects the service “Email” and thus all the users whose user agents have a client/server relationship with this specific server; however, other services (News, WWW, Chat) are still usable because they do not depend on a functioning Email service. We can conclude that inter-system dependencies are always confined to the components of the *same* service<sup>1</sup>.

Tracking inter-system dependencies is a particularly hard problem and, therefore, has seldom been done in practice, because making these dependencies explicit mandates that a server “knows” its clients, which is generally not the case.

However, it may be feasible for a specific set of services to compute the actual clients that are dependent on a specific server. This involves the gathering of information

<sup>1</sup>This is consistent with the widely accepted definitions of a networked service, e.g., in the ISO/OSI reference model.

- often from multiple places (with sometimes high costs in terms of required bandwidth, storage and processing time),
- through mechanisms specific to the service implementation and, often, proprietary w.r.t. the operating system.

To sum up, it is safe to say that while it is technically feasible to compute inter-system dependencies, there is a trade-off between the usefulness of the information and the potentially prohibitive costs of obtaining this information. We can also state that the computation of an all-encompassing dependency model that reflects the current state of service configurations is hard to achieve due to the high amount and the dynamics of runtime dependencies. A centralized storage of an instantiated model (e.g., in a management platform) is therefore prohibitively expensive. We discuss this in greater detail and present an approach to solve this problem in section 4. This approach is built on the definition of two different kinds of dependency models:

1. A **Functional Model** that defines generic service (database service, name service, end-user application service etc.) dependencies and establishes the principal constraints to which the other models are bound.
2. 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 completes the amount of information provided by the Functional Model.

A complete picture of all the dimensions for classifying dependencies is presented in section 3. In the next section, we will analyze which of these requirements for dependencies are met by current approaches.

## 2.3 Related Work

For some time now, dependencies between components (and their discovery) have been the subject of several research and standardization initiatives:

The OpenGroup *Distributed Software Administration (XDSA)* specification [12] addresses the mechanisms for software distribution/deployment/installation and identifies three kinds of relationships: prerequisite, exerequisite, corequisite. It should be noted that XDSA does not deal with *instantiated* applications and services and therefore does not represent means of determining dependencies between components at runtime.

In the *Common Information Model (CIM)* [3], dependencies – being usually 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, System, Application and Distributed Application Performance (DAP) Schemas each define dependency types in terms of abstract classes but 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*. Furthermore, despite early work [10] and several research [2] and standardization efforts [9, 6] no satisfactory application management solutions are available on the marketplace. The main problem is that comprehensive application management demands a large amount of management information, thus posing an additional development effort on the application developers [7]. To sum up, only the emerging standards XDSA and CIM specify dependencies reasonably well 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.

In order that we may design a methodology for identifying and representing dependencies, it is important to identify the different types of dependencies that can play a key role in the different phases of the service and application lifecycle. In the next sections, we will therefore present a classification of dependencies that will help us to compute dependencies in distributed environments.

### 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 key properties (or characteristics) of a dependency as shown in figure 2. The coordinate values of the axes are discrete. A dependency, from the viewpoint of classification, has the following characteristics:

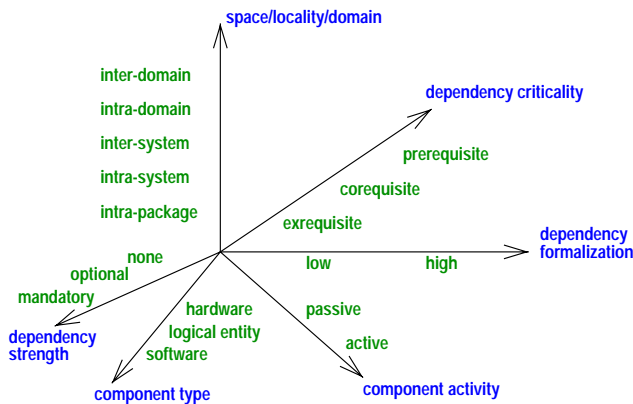


Figure 2: Multidimensional space of Dependencies

**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. When a dependency line crosses a domain boundary, this usually means that Service Level Agree-

ments are associated with this dependency. Thus when we are looking for dependencies that affect SLA management (i.e., to ensure the customer is obtaining the Quality of Service that was promised), inter-domain dependencies play a key role. On the other hand, a dependent service 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.

**Component Type** – what the antecedent component actually is, i.e., a piece of hardware, an end system, a software package, a service, etc. This distinction is important because different types of components tend to behave and fail differently.

**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 and must always have an “intermediary” that acts on behalf of it.

**Dependency Strength** – how strongly the dependent component depends on the antecedent resource. While it may seem useless to consider dependencies other than mandatory, it turns out that *strength* is a good metric to deal with intermittent dependencies. These reflect the situation where a component may require a resource only for certain periods of time: For an application working with a local Lotus Notes database copy, the Notes server is required only twice during its runtime, namely to perform initial and final replication of the database. The server may be unavailable during intervals in between without affecting the application adversely.

**Dependency Formalization** – what degree of formalization this dependency has and, thus, to which degree it can be determined automatically. This serves as a metric that helps to evaluate how expensive and/or difficult it is to acquire and identify this dependency, represent it, and, track this dependency during the lifetime of the component. For a “formalized” dependency, the cost of dealing with it is obviously lower than that for a “non-automated” or “not-well-formalized” one.

**Dependency Criticality** – how should this dependency be satisfied, in terms of availability of the resource/service this component depends on. It is important to note that as the dependency evolves with time, the meaning and the semantics of the three possible values change.

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 e.g., 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 that were not there before – such as the need for remote components which matters when the application runs, but should (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 life-cycle moves past installation. Another example (dependency upon DNS) appears only after the installation is complete and possibly after the configuration stage is done.

It is useful to monitor and study how the dependency characteristics change and how dependencies are dropped or added, as the dependency moves along the time line.

**Dependency Lifecycle:** It is useful to distinguish between *functional* and *structural* dependencies. 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 detailed information and is typically captured first at deployment or installation time.

## 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. Abstract service models have been used to address this issue [8], but this approach requires the collection and mining of large amounts of data. In the following sections, we will describe a pragmatic approach for dependency enumeration that can be realized on a variety of widely deployed operating systems *without* requiring detailed management instrumentation of the applications and networked services.

### 4.1 System Information Repositories

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, our approach is applicable to other repositories as well.

Usually, these 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.
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 a specific service are explicitly listed in the system repository. Section 4.2 will give an example of such a data structure.

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 captures dynamic information related to concrete service implementations. In the next sections, we illustrate our approach by means of an example.

### 4.2 Intra-System Dependencies

The *Object Data Manager (ODM)* is the repository of the AIX operating system and contains information about the different components of the operating system such as device drivers, networking services and graphical user interfaces, middleware (like object request brokers, message-queuing systems), development tools (compilers, debuggers, CASE-Tools) and additional components such as database and web servers. Furthermore, applications such as web browsers,

database clients, management platforms, groupware systems are also registered in the repository.

ODM obtains knowledge about new software components at their installation time from the `installp` software installation routine that is invoked by SMIT. This routine reads the software description template (an example of this simple template common to all applications is depicted below) that comes with the new software package, verifies the prerequisites and – upon successful completion of the prerequisite tests – installs the software and enters the description template into ODM. Note that the information contained in the template can be added after the application has been compiled. Thus, it does not necessarily have to be provided by the application developer (although this is desirable) but can be added by the system administrator or a third party.

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<sup>2</sup> (`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 indicates 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.

The query operations of the ODM API [4], a C-API containing 22 subroutines for various administrative functions, allow the generation of both **functional and structural service dependency models** at runtime. Figure 3 represents an extract of the functional service dependency model (including applications such as DB2 database clients, services such as NFS and DNS) of a running system and *not an abstract model*. It is built by evaluating the `description` and `prereq` fields of the above template. The structural

<sup>2</sup>The term "LPP" in the template stands for "licensed program product" and denotes a service component.

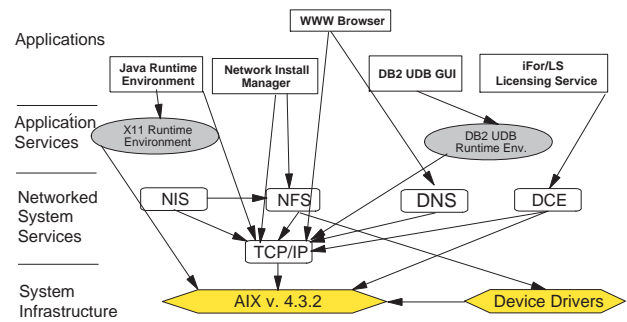


Figure 3: Extract of functional service dependency model

dependency model is much more detailed and omitted for the sake of brevity; it contains information on the different components that make up a service and is based on the `lpp_name`, `ver` and `rel` entries in addition to both fields cited above.

This analysis shows 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. The only requirement is that the application components be described using the above template. However, it should be noted that:

System repositories refer to *individual systems only* and thus yield, for the most part, Intra-system dependencies. Since end-to-end management mainly deals with Inter-system dependencies, we have to develop mechanisms that allow us to track the dependencies between client and server components located, in general, on different systems. This topic is addressed in the following section.

It is necessary to link the **static** information contained in the repository to **dynamic** information obtained from the processes at runtime (e.g., their state and priority, the percentage of used CPU cycles and memory, the userID of their owner etc.). A mandatory requirement is that the templates also contain the process names of their components (which is not the case in the current ODM implementation). However, as the process names are already known at installation time, they can easily be included into ODM by adding an additional item `"process name"` to the data structures.

### 4.3 Inter-System Dependencies

It is particularly important for end-to-end problem determination that information about Inter-system dependencies is available, because it is necessary to determine if the client *and* the server parts of a given service work properly. We will first examine the possibilities of determining problems on the client side and then describe what mechanisms are available for servers.

Very often, information about the servers to which a client connects is explicitly listed in the client configuration files: A DNS resolver not only has information regarding its do-

main name but also its name servers. NFS clients - in turn - know from which servers they need to mount file systems. Some counterexamples to this are the WWW and FTP services, where the server names are obviously not known in advance. However, this problem is alleviated by the fact that usually a proxy server is known, thus allowing one to determine if the problem is local to the client or lies beyond the proxy.

On the server side, we are able to make use of the dependencies listed in ODM also for the determination of Inter-system dependencies for the following reasons:

- Client and servers usually share the same functional dependencies (e.g., both web client and web server depend on the availability of the DNS service which - in turn - depends on a working IP service). The dependency model in figure 3 is valid for clients and servers.
- From an installation point of view, clients of a given service are prerequisites for their server counterparts. This implies that server software can only be installed if the client part is already present on the system; the ODM template for a TCP/IP server described in section 4.2 clearly states this in the second argument of the `prereq` attribute. We can therefore assume that the clients needed to test the servers are locally installed thus facilitating the testing of servers because no network and intermediate systems are involved.

Although we have seen that system repositories provide a considerable amount of information useful for application management, it should be pointed out that one specific kind of dependency cannot be tracked by them: it is yet not possible to determine if a software package is installed on a local or remote file system. The additional dependencies introduced by networked file systems (and their underlying operating systems) do not appear in the model.

Another issue is that the information identified in this section is relevant for configuration and fault management but not suitable for performance and accounting management: It is possible to verify whether a service and its prerequisites and peers are working but we cannot ascertain whether the services perform their duties in an acceptable time period. In order to satisfy performance-related SLAs, the applications need to be instrumented appropriately. The *Application Response Management (ARM) API* [1] with its recent extensions [5] is an example for such an approach.

## 5 Conclusion and Outlook

A key requirement for application and 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). We show that standard operating systems, such as WindowsNT, AIX and Linux, contain a wealth of information in their repositories that can be exploited for the *automated generation of dependencies*. Our approach has an added advantage in that it does not place the burden on application developers to instrument their applications.

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.

## Acknowledgment

The authors are indebted to Seraphin B. Calo and Nagui Halim for valuable suggestions and discussions.

## References

- [1] Application Response Management. Version 2.0, Tivoli Systems, November 1997.
- [2] P. Brusil, J. Hellerstein, and H. Lutfiyya. Applications Management – Current Practices, Research Results, and Future Directions. *Journal of Network and Systems Management*, 6(3):361 – 366, September 1998.
- [3] Common Information Model (CIM) Version 2.2. Specification, Distributed Management Task Force, June 1999.
- [4] IBM Corporation. *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*, October 1997. Chapter 17: Object Data Manager (ODM).
- [5] M. W. Johnson and S. Smead. *Beyond ARM 2.0 - API Extensions that enable pervasive Service Level Instrumentation*. Computer Measurement Working Group, December 1998.
- [6] C. Kalbfleisch, C. Krupczak, R. Presuhn, and J. Saperia. Application Management MIB. RFC 2564, IETF, May 1999.
- [7] M. Katchabaw, S. Howard, H. Lutfiyya, and A. Marshall. Making Distributed Applications Manageable through Instrumentation. *The Journal of Systems and Software*, (45), 1999.
- [8] A. Knobbe, D. van der Wallen, and L. Lewis. Experiments with Data Mining in Enterprise Management. In *Proceedings of the Sixth IFIP/IEEE Symposium on Integrated Network Management (IM'99)*, Boston, MA, USA, pages 353–366. IEEE Publishing, May 1999.
- [9] C. Krupczak and J. Saperia. Definitions of System-Level Managed Objects for Applications. RFC 2287, IETF, February 1998.
- [10] R. Sturm and J. Weinstock. Application MIBs: Taming the Software Beast. *Data Communications*, November 1995.
- [11] D. Verma. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.
- [12] Systems Management: Distributed Software Administration. CAE Specification C701, The Open Group, January 1998.