Managing Dynamic Service Dependencies

Peer Hasselmeyer

IT Transfer Office, Darmstadt University of Technology,
Wilhelminenstr. 7, 64283 Darmstadt, Germany
`hasselmeyer@ito.tu-darmstadt.de`

We anticipate that software development will be service-centric in the near future. Applications will be created from existing services that are distributed throughout a network. Sure enough, management of those components will be mandatory. While service management is usually service-specific, a few areas can be identified that can be addressed in a generic way. One of these areas is the management of dependencies between services. Despite its genericity, this problem is currently mostly addressed in a service-specific way. This paper details the need for a generic dependency management approach, identifies the key properties of dependencies as well as the requirements for dependency management schemes, describes the model derived from the requirements, and presents an implementation using the Jini connection technology.

Keywords: Dependency management, service management, component-based management, dynamic service networks

1 Motivation

Service management is becoming increasingly important. It is well established in the area of network services, such as web servers. With the migration to building applications from distributed services, service management will become even more important. While service management is mostly service-specific, there are a few areas that are important to all services. Besides state management [CCI92], an area of major importance is dependency management. Dependency management is a special area of relationship management as dependencies are a specific type of relationship. A component is said to *depend* on a service if it (maybe only temporarily) needs the functionality of that service. Although there might be dependencies between objects within a component ("intra-component"), this paper focuses on dependencies between components ("inter-component").

Dependencies have a certain set of properties. An important property is that they are dynamic and can change over the lifetime of the dependency. This is due to the fact that in a complex system of distributed components, components can become unavailable, may migrate, or may be upgraded. In these cases, connections between services and its clients must be altered to adapt to these changes.

The knowledge of a service's dependencies is important for a number of management activities. Fault management needs this information to track problems in a distributed service network. Configuration management needs this information to know which services are currently in use and appropriately adapt to changes in the environment. Accounting management needs to know dependencies to appropriately charge for service access. Policy-based management needs to know dependencies and must be able to change them to enforce the policies. All these management activities must have ways to learn the current dependencies, discover their properties, and possibly perform rebinding of services. As dependencies have a limited set of generic properties (described in Section 2), it is possible to represent and manage dependencies in a way that is not specific to a certain service. Furthermore, dependency information has to be made visible to the outside of the service. We will argue that it is mandatory that the service itself publishes and maintains its dependency data. Appropriate instrumentation of services is therefore required.

This paper deals with dependencies between *components* of a distributed system. Some of these components offer their functionality to other components. These are referred to as *services*. Although components accessing a service are said to act in the client role, they can be services themselves. Components can therefore form arbitrary usage structures which are usually not hierarchical. Viewed from one single component, the component will be the central node of a directed graph of components. Viewed together, components form a mesh of interconnected nodes that work together in a peer-to-peer fashion.

The paper is structured in the following way: Section 2 details the description of dependencies. Section 3 analyzes the requirements that a management scheme for dependencies has to fulfill. Section 4 presents our dependency management implementation using the Jini connection technology. Related work is reviewed in Section 5, while Section 6 concludes the paper and points towards areas of future work.

2 Dependencies

This chapter takes a detailed look at dependencies between software components and identifies their properties.

A dependency in the context of this paper is a special kind of relationship between two (or more) distributed software components. If a component needs a certain external functionality, it is said to depend on that functionality. In our model, functionality is offered by services. As a component is the unit of granularity, we only deal with dependencies between components. We do not address intra-component dependencies which relate software parts within a component. These are usually only important at development and compile time and are therefore a debugging topic. Some of these intra-service dependencies, e.g. the selection of a cipher suite, are nevertheless dynamic and might be configurable at installation or run time. As this type of configuration is almost always service-specific, it is not addressed in this paper.

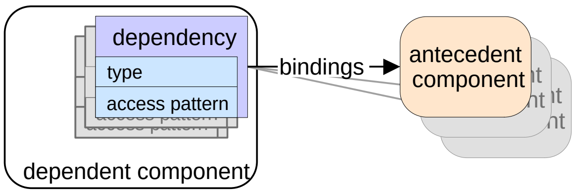


Fig. 1: Anatomy of a Dependency

A dependency is a directed relationship between a set of components. It has two roles: the *dependent component* and the *antecedent* (“free”) *component(s)* (see Figure 1). The cardinality of the dependent role, i.e., the number of components assigned to the client role, is exactly one. If there was no dependent object, the dependency would not exist. If more than one component depends on the same service, each dependent component has its own dependency. The maximum cardinality of the antecedent role can be one or infinite, depending on the application. If a word processor depends on a spell checking service, the maximum cardinality of this dependency is one. If a travel service depends on airline companies, it probably depends on more than one. The maximum cardinality of this dependency is therefore infinite. The actual cardinality of the antecedent role can change during the lifetime of a dependency and can be any value between zero and the maximum cardinality.

As indicated before, a dependency has an abstract *type* (previously referred to as “functionality”) and concrete *bindings* (the connections to service instances). The type describes what kind of functionality is required. A binding is a concrete association between the dependent component and an antecedent service. Bound services have to be of the type of the dependency.

When we analyzed the dependencies of a number of services and components that we implemented [ADH$^+$99, HSV00], we discovered that services are selected and accessed in a number of different *access patterns*. The discovered access patterns and their properties are listed in Figure 2. An access pattern can be described by a number of dimensions. The dimensions are: quantity of remembered services, quantity of accessed services, and the selection scheme. Furthermore, the access pattern depends on whether all services are the same or different from each other. This last property has to be distinguished from the type of the services. Although the services have the same type, they can differ in other respects, as mentioned before

name	quantity of remembered services	quantity of accessed services	services have priorities	services are
single	one	one	n/a	equal
random, round-robin	multiple	one	no	equal
ordered	multiple	one	yes	equal
first match	multiple	subset	yes	different
all	multiple	all	no	different

Fig. 2: Access Patterns

when we described the maximum cardinality: shopping services for example differ by the prices they charge for their products. All spell checkers should return the same results, though. This information is implied by the other data and is shown in the figure only for illustrative purposes. The quantity of remembered services can either be single or multiple. A word processor only needs to remember the single spell checking service instance it is currently relying on. The name resolver service probably remembers more than one domain name system (DNS) service instance for fault tolerance reasons. The quantity of accessed services can either be a single instance, a subset of the remembered services, or all remembered services. Again, a word processor only needs one instance of a spell checking service. A shopping application would use all services, if it wants to find the best deal available. Only a subset of the remembered services is needed if not all services offer the same data, but any answer will do. If you store configuration data distributed over a number of different services, the configuration from the first service that can supply a valid DHCP configuration will be selected. The selection scheme is only important if a subset (including one) of the remembered services is accessed. We found the following schemes used when only a single instance has to be selected: random, round-robin, and ordered. With the random method, one service instance is chosen at random each time it has to be accessed. The round-robin scheme also selects a different service instance each time, but in a sequential fashion (where the sequence is random but stays the same for all iterations). Both random and round-robin schemes can be used for load distribution. The ordered scheme assigns priorities to the available service instances. The service with the highest priority is always accessed first. In case it could not supply a valid result, e.g. because of a communications error, the service with the second highest priority is selected. The process continues until a valid result was returned. This last scheme is commonly used, for example, for DNS queries. It is similar to the first-match scheme but assumes that all services equal. The third dimension, the selection scheme, is currently not formalized. It is implied in the name of an access pattern and is therefore only useful for human operators. We found that the maximum cardinality is not an adequate property to describe a dependency. It conveys similar information as the quantity of accessed services but it is less detailed. We therefore refrain from using the maximum cardinality and only use the access pattern information instead.

The lifetime of a dependency equals the lifetime of the dependent component. Although a component might not immediately (or not at all) need the functionality of a service it depends on, the dependency is still there. An example is a print service that depends on an accounting service. Although accounting records are sent infrequently, the dependency exists over the whole lifetime of the print service. While the type (here: accounting service) and the access pattern of the dependency stay the same over the lifetime of the dependency, its binding is dynamic and can change. There might even be no bindings for periods of time. If we take a look at the example again, the accounting service might be replaced by a new one from a different vendor. In this case, the antecedent component changes and therefore the binding changes as well. In case the transition from the old to the new accounting service takes some time, there might be no bindings for some period of time. The print service might still be able to provide its service as it might be able to store accounting records until the accounting service becomes available again. Theoretically, the selection scheme of the access pattern could change over the lifetime of a dependency as well. The analyzed services did not show this behavior, though. As the selection scheme is chosen for important reasons (e.g. equality of service instances, quantity of required services, load distribution), we believe that the selection scheme will not change dynamically.

Because the lifetime of a dependency equals the lifetime of the associated dependent component and the dependent component of a dependency does not change over the dependency's lifetime, we see a dependency as a property of the dependent object. This is an obvious choice for dependency management, as each component knows best which services it relies on. This is in contrast to other relationships. In a connection relationship, the connected termination points might change over the lifetime of the connection. The connection can not be attributed to one of the termination points.

As mentioned before, bindings of a dependency are dynamic. Bindings change because of stimuli coming from either inside the dependent component or from the outside. Stimuli from the inside can include failure of the old binding, and load distribution. Stimuli from the outside are the results of some service management actions. The stimuli can have many sources. A few examples are service migration, service updates, load distribution, and changes in other services or policies. Changing the bindings from outside the dependent component is a configuration management activity as it alters the structure of the managed system.

To sum up this section, we list the properties to describe a dependency:

- the dependent component,
- the dependency's type,
- the access pattern, and
- the bound antecedent components.

3 Requirements for Dependency Management

Dependency management requires the infrastructure and participating components to fulfill a number of requirements. These are introduced in this section.

To perform dependency management, the dependencies must be accessible to management applications. For analyzing or displaying component dependencies, the available components and their dependencies must be visible from the outside. The data to be made visible is all the data identified in Section 2: the dependent component, the type of the dependency, the access pattern, and the current bindings.

In addition to retrieving the current dependencies, dependencies must be changeable at runtime—from within the component as well as from the outside. As mentioned before, the dependent object, the type, and the access pattern of a dependency stay the same over the lifetime of the dependency. The only data that can be changed is the binding of a dependency, i.e., the service instances a component uses. A component must have the possibility to change its own bindings as it must be able to respond adequately to services becoming unreachable. External components also must be able to change a components bindings. This might be due to a policy forbidding the use of certain services or an administrator directing traffic to a specific service instance. A dependency management framework therefore has to offer the possibility to change dependency bindings at runtime. Changes from within as well as from the outside the dependent component must be possible. It is important to note that changes are reconfigurations of the managed system. As changes—and therefore reconfigurations—can be performed by the component itself, it can be considered part of the managing system.

We already mentioned that we see dependencies as properties attached to dependent components. Furthermore, all dependency data should be supplied directly by the component having the dependency. This approach removes the possibility of having inconsistencies between dependency data and actual component interaction. Inconsistencies are possible for example in a scheme where dependency data is derived from configuration files. Many services only read their configuration data when specifically instructed to do so. Therefore, changing a configuration file is not enough to change the dependencies of the running service instance. In this case, dependency data derived from configuration files is inconsistent with actual service bindings—at least for a certain period of time. We therefore propose that dependency data be published by and directly retrieved from the running service instance. This data precisely represents the current component dependencies.

Changes in the dependency graph can be discovered by periodically polling the observed dependencies. Depending on the polling frequency and the number of observed dependencies, this method can waste a

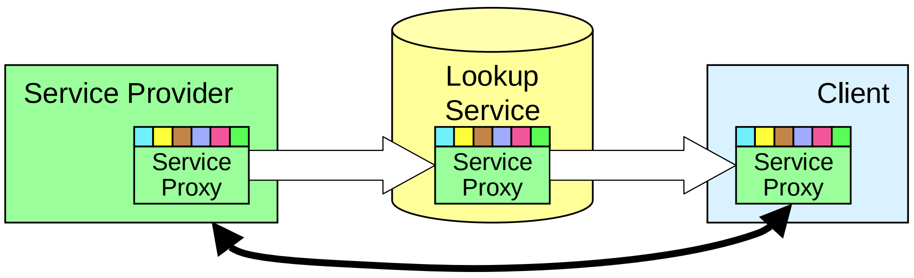


Fig. 3: Jini Architecture

large amount of bandwidth. A push model is therefore desirable. Appropriate notifications should be sent out upon dependency changes.

Bindings are references to running service instances. Service instances must therefore be identifiable, i.e., they must have a unique identifier. Appropriate identifiers are usually available in middleware platforms, e.g. distinguished names in the TMN framework, or object references in CORBA. It is important to note that this last requirement does not have to be fulfilled by dependencies but by the underlying service infrastructure.

To sum up this section, dependency management schemes have to satisfy the following requirements:

- dependencies must be visible from the outside,
- dependencies must be changeable from the outside, and
- dependencies must be supplied by the dependent component.

4 Implementation

Based on our dependency management analysis we implemented our own dependency management system using the Jini connection technology [Sun00]. This technology was chosen because it is a service-based infrastructure and contains a large number of features that we believe will be incorporated in future middleware architectures for service-based networks. In this section, we first introduce the properties of Jini that are relevant to this paper. We then describe the mapping of the discovered properties and requirements to the Jini technology and present a number of details of our implementation.

4.1 Jini

Sun Microsystems' Jini is a Java technology which allows services and clients on a network find each other in an easy, automatic, and dynamic way. This description is not intended to present an overview of Jini, it rather focuses on the features which are needed to understand the architecture described in this section: service proxies, attributes, and the role of the lookup service.

A Jini *federation* is the collection of all Jini-enabled components that take part in a Jini system at a certain point in time. This includes entities in the role of servers as well as in the role of clients. The bindings between these entities are established at runtime and can be changed dynamically during the lifetime of the components. The dynamic behavior of a Jini federation is enabled by the use of so-called *lookup services*. Services that want to offer their functionality to a Jini federation contact one or more lookup services and register with them. Clients also contact lookup services and ask for desired services. If a client finds more than one instance of a service type, it has to select the instance that it wants to use. Selecting a service instance relates to binding that instance into a dependency in our model.

Figure 3 shows a more detailed view of the actual interaction. It is important to note that all communication is mediated by *service proxies*. Proxy objects are supplied by the *service provider* that they represent. The service provider is the back-end and usually provides the actual service. The service proxy consists of a set of mobile objects which are sent to a client to interface with the service provider. A registration with a lookup service consists of the service proxy and a set of attributes called *entries*. Both parts of the registration are stored at the lookup service for later retrieval by service users. The entries as well as the service proxy are arbitrary Java objects. As Jini builds on the code mobility capabilities of Java, these objects can travel around the network and do not only contain data but also code.

4.2 Dependency Management Architecture

Dependencies consist of the following data: the dependent component, the type, the access pattern, and the antecedent services. Appropriate storing places have to be allocated for this data. As we consider dependencies properties of the dependent service, it was an obvious choice to attach dependencies to the component they belong to. The information about the dependent component is therefore implicitly stored in this attachment. The remaining three properties are accessible via objects that implement the `Dependency` interface (further described in Section 4.3).

Attaching dependency data to services is possible in many ways. We evaluated two possibilities: entries and dedicated objects. As described before, entries are arbitrary objects that are attached to service proxies. Dependencies could be stored as entries in the lookup service. An advantage of this method is that entries are searchable. Management applications therefore need to contact only a small number of services (all reachable lookup services) to discover the dependency graph. They can also ask lookup services for components that depend on a certain type or even a certain instance of a service. Service maps could be created quickly and without a large communications overhead. There are a number of disadvantages, though. First, this method increases the memory usage of the lookup service as well as of all clients. As entries are attached to services, they are downloaded by every client that accesses the service. Even if the client is not interested in management data, it will receive all the dependency information. Furthermore, this approach does not completely fulfill our requirement that dependency data has to be supplied directly by the component. To keep the dependency information up-to-date, dependency data has to be sent to the lookup service. As pushing the data to the lookup service requires remote communication and therefore takes some time, dependency data might be out of sync with the actual service bindings.

Because of the mentioned drawbacks of storing dependency data in entries, we chose a different method. The method is based on the normal Jini management pattern and uses dedicated management objects. Manageable services implement the `Administrable` interface. The interface contains a single function that returns a dedicated management object. The dedicated management object grants access to the dependencies. In contrast to the first solution, this approach achieves separation of concerns (between the “real” service and the management functions) and does not force regular clients to download dependency management data. The requirement of dependency data being directly supplied by the component is fulfilled because access to dependency management data is always channeled through the management object which belongs to the service. The main disadvantage of this scheme is that dependency data is not available at a single location. As each component is responsible for its own dependency data, dependency management data is distributed over all participating components. This achieves scalability, but requires a large number of components to be contacted if a complete picture of the current dependencies is needed. Furthermore, as dependencies are directed relationships, finding all components that depend on a given service requires all components to be queried for their dependencies.

In a Jini federation, usually only services are registered with the lookup service. Components which only act in the client role are not registered. They just contact the lookup service if they need the help of some service, i.e., if they depend on some service. To make our system work, all components—including clients—have to be visible and therefore registered with the lookup service. This makes sense as all components perform a service now, even components that used to be clients: they all offer the dependency management service now.

Identification of services in Jini is easy. A unique identifier is assigned to each service when it registers with a lookup service for the first time. A Jini service is required to make this identifier persistent so that it keeps the identifier even if it was stopped and restarted. The identifier even stays the same if the service migrated. The identifier is therefore location independent.

```
public interface Dependency extends java.rmi.Remote {
    public Class getType() throws java.rmi.RemoteException;
    public AccessPattern getAccessPattern() throws java.rmi.RemoteException;
    public ServiceID[] getBindings() throws java.rmi.RemoteException;
    public BindingRestriction getRestrictions() throws java.rmi.RemoteException;
}

public interface ManagedDependency extends Dependency {
    public void setRestrictions(BindingRestriction limits)
        throws IllegalServiceTypeException, SchemeMismatchException,
               java.rmi.RemoteException;
    public ServiceID[] getAvailable() throws java.rmi.RemoteException;
}
```

Fig. 4: The `Dependency` and `ManagedDependency` Interfaces

4.3 Implementation Details

To demonstrate the viability and usability of our dependency management architecture, we implemented both a manager as well as number of managed components. The components were already available from previous work [HSV00], but they did not have a dependency management interface. We therefore only added dependency management instrumentation to our components. This is especially interesting as it shows how much work is involved when “upgrading” existing components.

Figure 4 shows the dependency interfaces. We separated the adjustment functionality from the read-only access to allow for dumb components that do not offer restricting service bindings. We discourage the use of read-only access, though. `Dependency` and `ManagedDependency` are interfaces and not classes because different components might want to use different methods of determining a dependency’s attributes. The interfaces are remote because objects implementing the interfaces are most likely accessed remotely. As type and access pattern cannot be changed, they can only be read. Actual bindings can only be read as well, as they are the sole responsibility of the service. Nevertheless, they can be influenced by specifying restrictions on the possible bindings. These restrictions can either be inclusive or exclusive, i.e., they contain a list of services to be used or not to be used. As every component can have a different view of the set of available services, the services available to the administered component can be retrieved.

As mentioned before, dependencies can be accessed via dedicated management objects. A management object that is able to execute dependency management functions has to implement the `DependencyMan-agement` interface which consists of only a single function to retrieve the service’s dependencies. The way to get access to the dependencies of a service is depicted in Figure 5.

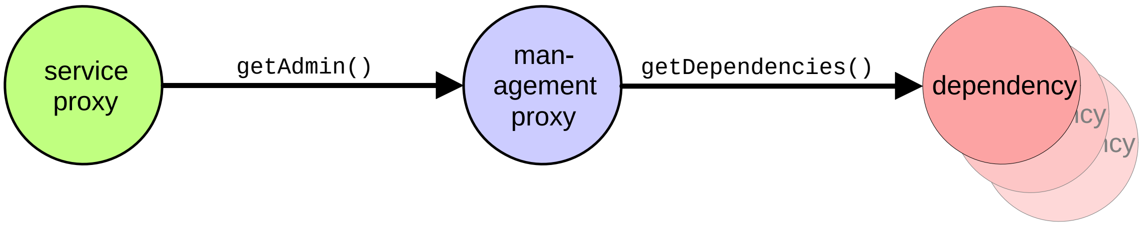


Fig. 5: Getting Access to Dependencies

To manage our dependency management-enabled components, we enhanced our service browser with the desired capabilities. In addition to displaying the current set of components, we can display the complete dependency graph. An example is shown in Figure 6. As this “global” view becomes complex with a growing number of services, we developed a hierarchical presentation, shown in Figure 7. For this view a single component is selected which becomes the root node of the displayed graph. Below, all dependencies of the root component are shown. This method goes on recursively until all dependencies are displayed. Dependencies can be further examined by clicking on the dependent service. All services of the type of the dependency will be shown and the administrator can alter the binding restrictions.

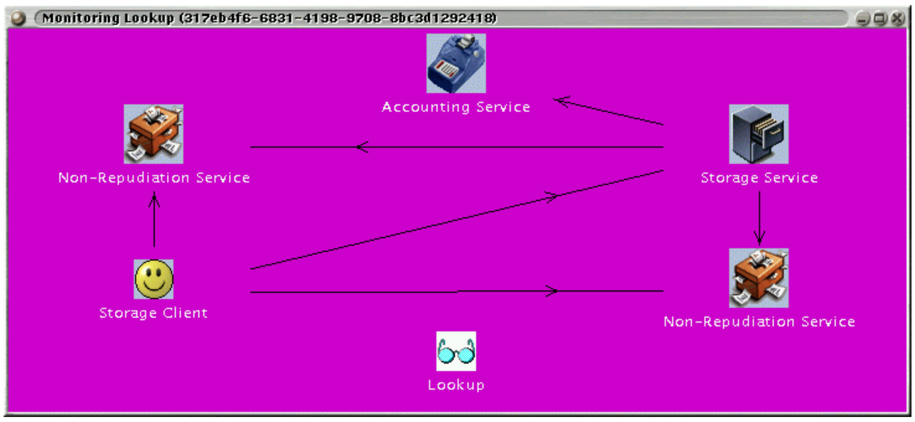
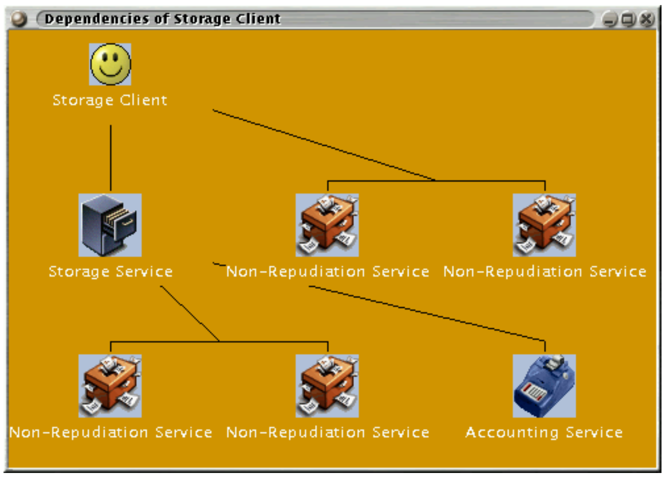
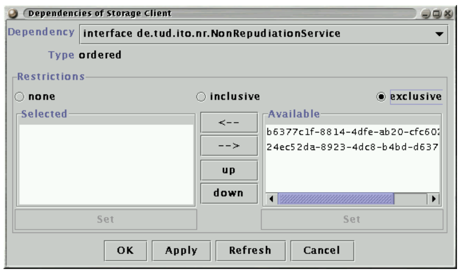


Fig. 6: "Global" View of Service Dependencies




Dependencies of Storage Client

Dependency: interface de.tud.ito.nr.NonRepudiationService

Type: ordered

Restrictions: none [✓ exclusive] inclusive

Selected:

Available:
b6377c1f-8814-4dfe-ab20-cfc60
24ec52da-8923-4dc8-b4bd-d637

Fig. 7: Dependency View of a Particular Component

Enhancing the existing components with dependency management facilities was surprisingly easy. Three steps were involved: a dedicated management object had to be added to the service proxy, the appropriate dependency object had to be set up, and the binding restrictions had to be enforced. While the first two steps are straight forward, the last step requires more work. Our goal was to leave the existing code in its original state as much as possible. As the components use convenience objects for caching service references, we simply had to add the required service filtering. This was easy, as the convenience objects already support filtering which was not used before in the clients.

5 Related Work

[KC00] discusses dependency management in component-based distributed systems. Jini can be seen as an instance of such systems. The paper does not address the topics presented in this paper, though. It rather focuses on software packages, their installation, and their interdependence. While software installations face some of the same problems as component-based services, they are usually more restricted. In particular, they do not have the possibility of being installed or started without all dependencies satisfied. Furthermore, dependencies between different software packages can usually not be changed while the system is running. Therefore, changing dependencies at runtime by an administrator is not an issue here.

Monitoring the interaction of distributed components is discussed in [DBZvS00]. The software development aspect of cooperating distributed components is explored. Monitoring communication between components by using CORBA interceptors and its application to finding faults is described. We designed and implemented a similar system using the Jini connection technology [HV01]. As these schemes are aimed at debugging distributed applications, dependencies can be monitored, but not changed.

[KK00] discusses dynamic dependencies in depth. Although their work is similar to the work presented in this paper, their architecture and implementation differs in a number of areas. They are addressing dependencies between services that are not instrumented. They rather extract dependency information from configuration files and software installation databases. As we argued in the course of this paper, up-to-date dependency information can only be supplied by the running service itself. Furthermore, it is not clear how the dynamism of dependencies is handled in their work.

A number of standardization bodies worked on relationships and dependencies. In the work of the Tele-Management Forum [Tel00], dependencies are identified but not further studied. They included the requirement that the dependent object must publish its dependencies, just as we did in our work. The General Relationship Model (GRM) [IT95] describes an architecture for arbitrary relationships as well as methods to define and represent them in the TMN framework. The model does not specifically address service dependencies. These could be modeled with the methods of the GRM, though. The CIM model [Dis99] defines dependencies between services, but it does not capture the dynamism of service dependencies. CIM service dependencies are rather requirements on the existence of services and the order of their execution. They are not supposed to change during the lifetime of the dependency. As none of the models is aimed at dynamic service dependencies, none of them details dependencies with access patterns.

6 Conclusion

This paper analyzed dynamic dependencies between distributed software components. The properties of the dependencies were described. Requirements for service dependency management schemes were identified. Based on the dependency analysis and the described requirements, a dependency management scheme was designed and implemented using an infrastructure for distributed cooperating components. Although the system fulfills its requirements, some further research has to be carried out.

As mentioned before, our architecture does currently not allow a "reverse" dependency tree to be built easily. All components have to be contacted and all their dependencies have to be analyzed. We are therefore thinking about an additional requirement that forces dependent objects to register with antecedent objects. Antecedent objects would therefore know who is currently using them. This information could be made public and used by management applications to easily create a reverse dependency tree.

References

[ADH$^+$99] Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer, Roger Kehr, and Andreas Zeidler. A Framework for the Integration of Legacy Devices into a Jini Management Federation. In *Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM '99)*, October 1999.

[CCI92] CCITT. *Recommendation X.731 (01/92) – Information technology – Open Systems Interconnection – Systems mnagement: State management function*. January 1992.

[DBZvS00] Nikolay K. Diakov, Harold J. Batteram, Hans Zandbelt, and Marten J. van Sinderen. Monitoring of Distributed Component Interactions. In *Workshop on Reflective Middleware (RM 2000)*, April 2000.

[Dis99] Distributed Management Task Force. *Common Information Model (CIM) Specification, Version 2.2*, June 1999.

[HSV00] Peer Hasselmeyer, Markus Schumacher, and Marco Voß. Pay as You Go – Associating Costs with Jini Leases. In *4th International Enterprise Distributed Object Computing Conf erence (EDOC 2000)*, pages 48–57. IEEE Publishing, September 2000.

[HV01] Peer Hasselmeyer and Marco Voß. Monitoring Component Interaction in Jini Federations. In *To appear: Symposium on The Convergence of Information Technology and Communications (ITCom 2001)*, August 2001.

[IT95] ITU-T. *Recommendation X.725 (11/95) – Information technology – Open Systems Interconnection – Structure of management information: General relationship model*. November 1995.

[KC00] F. Kon. and R. H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26–36, January 2000.

[KK00] Alexander Keller and Gautam Kar. Dynamic Dependencies in Application Service Management. In *2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.

[Sun00] Sun Microsystems, Inc. *Jini Architecture Specification – Version 1.1*, October 2000.

[Tel00] TeleManagement Forum. *Generic Requirements for Telecommunications Management Building Blocks (GB909 Part1)*, June 2000.