An Infrastructure for the Management of Dynamic Service Networks

Peer Hasselmeyer
Darmstadt University of Technology

Keywords: Service management, component-based management, distributed management, management frameworks, service composition

1 Abstract

Recently, infrastructures for creating and deploying dynamically cooperating distributed services became popular. An area that these infrastructures address only insufficiently is the administration of services and their relationships. This article takes an in-depth look at some management topics that are particularly important in a dynamic service environment: service lifecycle and service dependencies. It states requirements for these areas that a management system has to fulfill. The article also describes MADYSON, a management infrastructure that was developed based on these requirements.

2 Motivation

In recent years, one can observe a trend towards a new software development paradigm. Applications are not monolithic blocks anymore. Rather, they are composed of independent distributed components that dynamically cooperate to achieve common goals. A few architectures fuel this trend. The most prominent ones are Web Services [1], Jini [2], and e-speak [3]. Although these technologies have differing goals, they share a number of basic principles. They are service-oriented, cater to dynamism in the set of services, and foster the cooperation of services. Because of these properties I collectively refer to these technologies as “dynamic service network” technologies.

Following a service-oriented approach offers a number of advantages. It allows the dynamic binding of components at runtime to build loosely-coupled applications. Components of the system can be exchanged at runtime without the need to take down other components or even the system as a whole. If a component of the system becomes unreachable by, e.g., a server-crash or a broken network connection, another component offering the same functionality is used instead.

An area that is not or only insufficiently addressed by the current architectures is the management of services and their relationships. It is only the first step to have an architecture that allows services to cooperate. Another step is to be able to manage the services so that they can perform their tasks effectively and efficiently. It is not enough that a component rebinds to a different service in case of a communications error. More subtle errors might only be detectable by analyzing the communication relationships between components.

In the course of this article I identify a set of requirements for managing components in a dynamic environment. The requirements include topics in the areas of service lifecycle and dependency management. To show how these requirements can be met, I also describe a matching management architecture and its implementation.

3 Dynamic Service Networks

The term *service* as used in this article describes an entity in a distributed system that offers certain functionality to other entities of that system. Services are usually pieces of software, possibly embedded in hardware devices. To provide their functionality, services might make use of other services.

In addition to services, the distributed system might also contain clients that only access services but do not offer any functions to other entities. All entities (services and clients) are collectively called *components*. The functions a service offers are usually described in some way, e.g. with the extensible markup language (XML) or as a Java interface. I refer to these descriptions as *service interfaces*.

Services can be as small as random number generators, but also as large as customer relationship management systems. The most appropriate size lies somewhere in-between. Good examples are credit card billing services and stock quote services.

Throughout the article the term *dynamic service network* (DSN) is used to describe a network of distributed components that cooperate to achieve common goals. The cooperation happens in a dynamic fashion: components use services only for a limited period of time. Connections between components can change over the lifetime of the components. The term "dynamic service network" refers to a particular instance of a network exhibiting such properties. The size of a DSN at a specific point in time is defined by its members. Members of a DSN are all those components that can reach other services of that particular network. Figure 1 shows an example of a dynamic service network. The stock analysis service shown in the Figure does not know the stock quotes by itself. It rather retrieves them from an appropriate service when needed. As shown in Figure 1, more than one stock quote provider might be available and a service can choose any one of them. Applications can interact with both services, depending on their functionality.

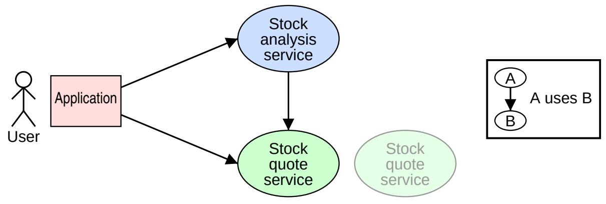

Figure 1: An example of a dynamic service network.

The most remarkable feature of DSNs is their dynamism. It appears in two different flavors. First, and most obvious, the connections between components are not static. They are not set at service creation time, not even at service installation time. Connections between components are established at runtime. And they can even change during the lifetime of components. Second, components themselves are not tied to a specific host. As they establish their connections at runtime, it does not matter where components are running, as long as they can interact with other components of the system. This dynamism makes deploying services tremendously flexible. As a side effect, it introduces new challenges for service creation as well as for service management. The implications for service management are discussed later in this article.

The reason for the dynamism in DSNs is the autonomy of the components. Autonomy refers to the fact that individual components operate independent of all other components. Components encapsulate their internal state and control who—and to what extent—gets access to their state and their services. Other components do usually not have direct influence on the state of a component. To the outside world, each component appears to autonomously control its own state and lifecycle. This implies that services can appear and vanish at any point in time. Components depending on a disappearing service must be able to deal with this situation appropriately and change over to a different service instance that provides the same functionality.

Although components appear to be autonomous, they usually operate under the control of a human administrator. Just like communication and computer networks, service networks need to be administered to ensure appropriate levels of performance, security, etc.

4 Related Work

There are different understandings of what a "service" denotes. The above definition of a service is rather broad and similar to the definition in [4] where a service is defined as a certain set of interactions. Other work is often restricted to certain kinds of services, especially data transport services. These restricted systems can not be used for the management of arbitrary (software) services. Their limitations are due to the intrinsic property of service management being mostly service-dependent. Nevertheless, a number of areas can be identified that can—and should—be handled in a service-independent manner. Work has been done on generic service management, but most of it is restricted to certain management functions.

The Telecommunications Information Networking Architecture (TINA) specification [5] includes some work on service management. The description is rather high-level and vague. It states a number of problems related to service management, including lifecycle, security, and dependencies, but it does not propose any solutions.

Sahai et al. describe requirements for the management of e-services in [6]. They only address performance management, though. The other functional management areas are not touched. Some of the special requirements introduced by dynamic interactions and autonomy are mentioned but not further explored. As performance management is mostly service-specific, infrastructure support is limited to the specification of relevant management information.

Another closely related area is component-based management systems. The TeleManagement Forum describes requirements for a comprehensive architecture in [7]. The document describes requirements for an architecture for component-based management systems. The managed entities are assumed to be regular network elements, though. The management of components, as analyzed in this article, is not considered. Management activities are therefore modeled in a host-oriented and not a service-oriented way that I advocate.

5 DSN Management

Management of DSNs builds upon "regular" service management and therefore has some of the same requirements and features. In addition, the dynamism and service-orientation of these networks has to be accounted for. I analyzed a number of service-based applications that my colleagues and I had implemented before. The analysis revealed a number of management areas that are affected by the properties of DSNs. These areas are service lifecycle, service dependencies, management model, security, and accounting. Only the first two topics are described in this article, as they are affected to the largest extent. It is important to note that the requirements described here are largely influenced by the desire to preserve the main properties of DSNs: dynamism and service-orientation.

Nowadays, service management is handled at the platform level. Usually, the platforms that host services, e.g. application servers, are managed, not the individual services directly. What follows is a platform-oriented view, although a service-oriented view seems more appropriate (see Figure 2). A service-oriented management view avoids the need to map between two disparate views: the platform view and the service view. For service management, the platform view is usually irrelevant.

The service view is more dynamic because services usually change at a faster rate than the hardware they run on. Service management applications should keep up with that pace. They should be as flexible and dynamic as the services that they manage [8]. I therefore advocate a

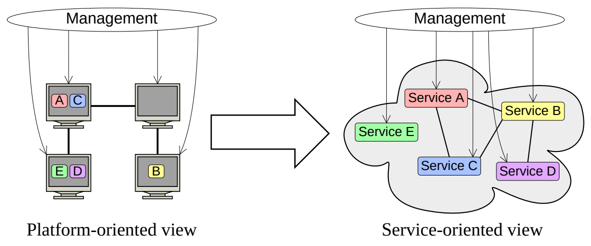

Figure 2: Platform-oriented versus service-oriented view.

management at the service level rather than on the platform level. The architecture described in this article tries to accomplish that goal.

5.1 Service Lifecycle

The lifecycle of a service includes the design, the implementation, the activation, the operation, and the withdrawal of a service. A management architecture is only concerned with the activities once a service is ready to be deployed. The design and implementation phases are therefore not covered in this article. The operational phase includes the following activities: installation, activation, deactivation, and removal. It might also include migration, replication, and updates. I do not address these activities separately as they can be modeled as special cases of the previous activities. For example, migration involves deactivating a service at the old host and activating it at the new one.

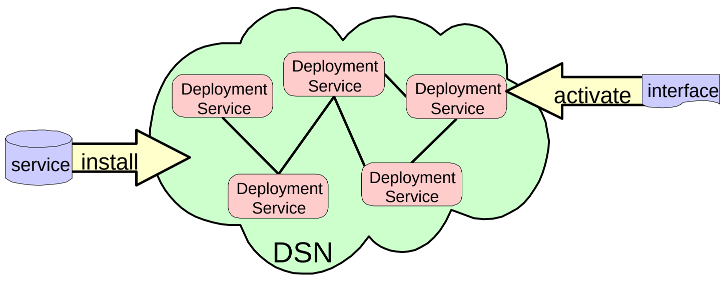

Figure 3: DSN Lifecycle Support.

As shown in Figure 3, installation is separated from activation. Installation is the act of introducing certain functionality (in the form of a service) to the distributed system. Removal withdraws a service's code from the system. As we have a service-oriented view where individual hosts are not relevant, both functions are not host-specific. They apply to the DSN as a whole. This is a fundamental difference to the common notion of installation where software is installed on a specific host.

Management applications must be able to install and remove services. Furthermore, a list of currently installed services must be accessible. Required information includes data identifying the code, e.g. the service's name and its vendor, version information, and the service interfaces provided by the code.

Activating a service means that the service's code is executed on a host of the distributed system. Only a subset of the connected hosts will be available for running arbitrary services. These hosts must be identifiable and administration applications must be able to start services on them. Although activation appears to be host-centric, it does not violate the service-oriented view. Hosts can simply be represented as services within a DSN.

Due to services being autonomous, deactivation, suspension, and resumption are handled by services themselves. These operations are therefore separated from service installation and activation. This method gives services the chance to perform specialized shut-down activities, such as waiting for ongoing transactions to finish. It also means that each service has its own management agent.

Just as for locating services, for the purpose of activation services are only identified by their interfaces. When activating a service at a certain host, the management application asks for a certain interface to be "started", not for a specific code bundle. The host starts one of the available code bundles that implement the given interface. Management applications do therefore not need to know anything about code bundles, only about interfaces. The administrator has to ensure that only desired code bundles are installed.

Most services need a certain set of configuration data. A service must be able to store and (on a restart) retrieve its settings. This is not a trivial issue as dynamic services can be started on arbitrary and varying hosts. As hosts may have different set-ups, the traditional solution of a configuration file at a well-known location is not viable. The infrastructure therefore has to provide some other means of handling configuration data in a location-independent way.

It is important to note that not all services can be freely migrated. Plain software services can be run on arbitrary hosts as long as there is enough computing and memory capacity. Some services require access to further facilities, though. An example is a private branch exchange (PBX) that is connected to a certain host. Unless the PBX is network-accessible, the service making the PBX available to the DSN has to run on the host that is connected to the PBX.

In summary, I discovered the following requirements for service lifecycle management:

- Management applications must have means to install, activate, terminate, and remove services.
- Management applications must have means to list installed service code bundles and running service instances.
- Services must be able to delay their termination.
- Management applications must have means to determine the state (e.g. suspended) of services.
- Management applications must have means to change the state of services.
- Service configurations must be storable and retrievable in a location-independent way.
- Each configuration must not be handed out to more than one service at each point in time.

5.2 Component Dependencies

A dependency is a directed relationship between a set of components. If a component needs access to a certain set of external functionality (i.e., a certain type of service), it is said to *depend* on that functionality. As shown in Figure 4, a dependency has two roles: the *dependent component* and the *antecedent* ("free") *component*. In each dependency, exactly one component assumes the dependent role, while an arbitrary number (including zero) of components may play the antecedent role. The actual cardinality of the antecedent role can change during the lifetime of a dependency.

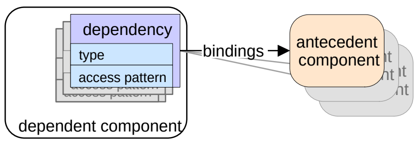

Figure 4: Anatomy of a Dependency.

A dependency has an abstract *type* and a number of concrete *bindings* [9]. The type describes what kind of functionality is required by the dependent component. It therefore refers to the interface that candidate services have to provide. The bindings are references to the antecedent services. Bound services have to be of the type of the dependency. The access pattern describes how the dependent component selects and accesses antecedent services, e.g. in a round-robin fashion. All data of a dependency is static except for the bindings. The bindings might change over the lifetime of the dependency. Changes can be triggered by external or internal stimuli and are the result of, e.g., service migration, service updates, load distribution, and changes in policies or other services.

For certain activities, management applications need to know a service's dependencies. One such activity is locating the root cause of a failure or a performance degradation. Also, service usage accounting could be based on dependency data.

For activities like failure locating, read access to the current dependencies is sufficient. Other activities like configuration management also need to change the connections between services. A facility to actively adjust dependencies from outside the service must therefore exist.

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 an external entity derives dependency data from configuration files. If service instances supply dependency data themselves, it precisely represents the current dependencies. In addition, changes to dependencies should only be performed by the dependent component. Management applications have to instruct the component to perform the change.

In summary, I discovered the following requirements for service dependency management:

- Management applications must have means to discover dependencies.
- Dependency data has to be supplied by dependent components.
- Dependency changes should be published via event notifications.
- Management applications must have means to change dependencies (bindings).
- Only dependent components should enact dependency changes.
- All components—services as well as clients—have to publish their dependencies.

5.3 Further Requirements

There are some more requirements for a system to be manageable. Most of them are well-known from traditional middleware. It is nevertheless important to state them explicitly. These requirements are:

- Components must be identifiable. As dependencies point to other services, references to services must be possible.
- Components must be discoverable at runtime. Managers must be able to detect the addition and removal of components, so that that they can automatically take care of new services.
- There must be a way to deliver event notifications that tell components about state changes of other services. Such a facility eliminates the need for constant polling and thereby helps to avoid congestion at management services.
- Distributed transactions are needed for a number of management activities.

6 Architecture

Taking the requirements stated before, I developed MADYSON (Management Architecture for Dynamic Service-Oriented Networks) that meets most of them. The requirements apply to the system as a whole. Any part of the system can therefore fulfill the requirements. In the case of service management, the system can be divided into three logical parts: the managers, the services, and the infrastructure that connects the components. To maximize reuse and to facilitate the development of management-enabled components, a goal of MADYSON is to place as much functionality into the infrastructure as possible. Nevertheless, the components need to cooperate to a certain extent.

MADYSON manages Jini services. Jini was chosen because it was the first middleware platform that allows distributed services to cooperate dynamically in an easy way. A Jini system is a loosely-coupled federation of service-providers and service-consumers. An entity can assume both roles at the same time. Connections between service-providers and consumers are established via the service registry, the so-called *Jini lookup service* (LUS). Services register with the LUS while clients find services by querying the LUS for appropriate entities. If more than one matching service is available, the client selects the one to use. Service lookups are interface-based. Clients do not ask for a specific service implementation, they rather look for a specific set of functionality described by a Java interface. Locating lookup services is done via multicast. The prior configuration of the registry's location is therefore not needed. In addition, Jini offers a number of facilities that ease the development of distributed applications. These facilities are event notifications, transactions, leases, and tuple spaces.

Although the services to be managed are all implemented using Jini, the management technology does not necessarily have to be Jini. It could be a different technology, e.g. the Simple Network Management Protocol (SNMP), the Common Object Request Broker Architecture (CORBA), or the Java Management Extensions (JMX). I nevertheless chose Jini and Java Remote Method Invocation (RMI) as the management technologies due to their relative ease of use. An example is Java's code migration facilities which make service deployment easy. For the prototype, the benefits of the other technologies, e.g. interoperability, did not seem to offset the increased complexity introduced by their use. In addition, the use of Jini allows following a service-oriented approach for management applications. The dynamism and service-orientation of DSN technologies thereby spread to management applications. Management applications are built from cooperating components—just like ordinary services. Also, there might be more than one manager per managed service.

Furthermore, the implementation of MADYSON makes use of the general middleware facilities that come with Jini. MADYSON relies on the Jini lookup service, the distributed events, and the leasing facilities. The lookup service not only serves as a registry for regular services, but for management interfaces as well. The management interface of a service is modeled as a Jini service which is completely separated from its regular service interface. Management applications can therefore administer services without knowing their regular interfaces.

6.1 Service Lifecycle

MADYSON's support for the service lifecycle consists of a number of services and helper classes. There are three service types: service repository services, deployment services, and configuration services. The architecture is depicted in Figure 5.

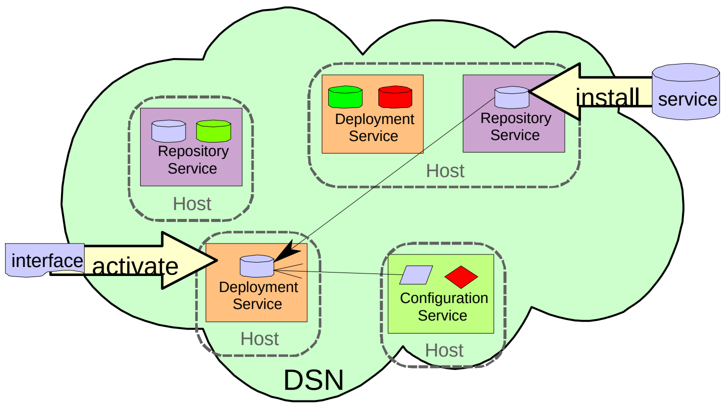

Figure 5: Lifecycle Architecture.

The repository service is a kind of database that stores and retrieves program code. Services can be introduced to a distributed system by adding the service's code and its description to the repository's database. In an analogous way the service can be removed from the system. The repository also allows listing the stored services as well as retrieving the program code from its database.

A deployment service runs arbitrary services. It retrieves service code on demand from a repository service. Service installation is therefore not host-specific. Adding a service's code to one repository service makes it available to all deployment services of that dynamic service network.

Services get their configurations from a configuration service. This service is similar to the repository service, but instead of storing program code it stores configurations. As services do not have any configuration information when they start up, they can not identify their particular configuration data. The solution used in MADYSON is based on the assumption that a service does not need to identify its own configuration. It only needs to access some configuration data bundle which it understands. Services therefore identify configurations by their type. They ask the configuration service for a configuration that conforms to a certain Java interface. If such a configuration is available, it is handed out to the service. In case no matching configuration is found, the service has to use its internal default values until an administrator configures it.

A single configuration should only be used by one service at a time. Configurations are leased and therefore handed out for a limited amount of time only. As long as a service needs the configuration data, it will extend the lease and be allowed to use the configuration. When a service is finished using a leased configuration (e.g. because it was stopped) the configuration is freed and can be used by another service. In case a lease taker forgets about a lease (e.g. because it crashed), the associated configuration is only blocked for the remaining lease time.

6.2 Component Dependencies

As a dependency has exactly one dependent component, MADYSON considers a dependency a part of the dependent component. Dependencies are therefore attached to the component they belong to. The remaining properties are accessible via dependency objects which can be obtained from the dependent service via a dedicated management interface. Access to dependency management data and functions is always channeled through the dependency objects. As these objects are a part of the dependent service, the requirement that services supply their own dependency data is met. As each component is responsible for its own dependency data, dependency management data is distributed over all participating components. This achieves scalability, but a large number of components must be contacted if a complete picture of the current dependencies is needed.

MADYSON contains interfaces that describe the attributes of and the access methods to dependencies. MADYSON also offers classes that implement these interfaces to facilitate the addition of dependency management functions.

A complete dependency graph includes the dependencies of clients. Clients must therefore make their dependencies available. Although this sounds like an easy effort, it actually means that clients become servers as well. They must be equipped with an interface that allows management applications to access their dependencies. The infrastructure contains classes to make this as easy as possible.

6.3 Further Requirements

Identification of services is possible with unique identifiers. Lookup services hand these identifiers out when services register for the first time. Services have to make them persistent, e.g. at the configuration service, so that they survive service restarts.

The Jini infrastructure also handles service discovery. Every service has to register with the available Jini lookup services. These notify managers of the appearance and removal of services.

Notifications and distributed transactions are inherent features of the Jini platform. Jini only provides the basic infrastructure, though. Services therefore have to supply notifications for state changes. They also have to provide appropriate functionality and interfaces to let them be part of a transaction.

7 Experiences

I implemented a prototype of the described architecture. The implementation consists of the infrastructure as well as a small demonstrator. The demonstrator is an extension of an existing service federation, a dynamic least cost routing (LCR) application [10]. The demonstrator shows a number of properties of the architecture:

- The lifecycle of services can be administered, e.g., another instance of the LCR service can be started on a different host when the existing one reaches its capacity.
- Dependencies can be managed, e.g., the LCR service can be restricted to use only certain communication providers.
- Configurations can be stored and retrieved by services from arbitrary locations.

The existing services as well as the client had to be management-enabled to cooperate with the infrastructure. Also, an appropriate manager had to be implemented. I discovered that it is usually rather easy and straightforward to add the desired management functionality. The reason is that MADYSON offers a number of helper classes that perform most of the work in common scenarios. Although special functions might be required under certain circumstances, I did not encounter a service that required enhancing the helper classes.

Making service dependencies administrable was one of the easiest jobs. All the existing services use helper classes to locate and bind to other services. As shown in Figure 6, these helper classes can be attached to peers from the management infrastructure that handle all management-related work. Only some meta data further describing the dependencies has to be added by hand.

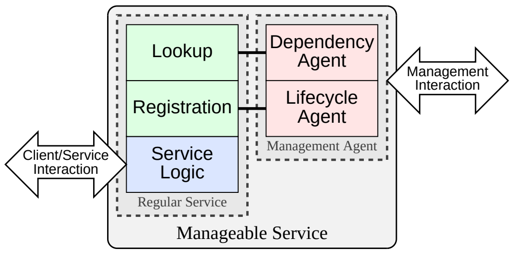

Figure 6: Integration of Helper Classes.

Making the service's lifecycle administrable was similarly easy. Letting services be started remotely required little to no work as services are started in the same way as regular Java applications (invoking `main()`). Stopping, suspending and resuming services was easy as well, as all services use some given helper classes that handle service registration. Again, peer objects from the management infrastructure attach to these helper classes and handle suspension, resumption, and shut-down. This method works well for simple services. More complex services might require specialized shut-down procedures. They might, for example, have to wait for the end of ongoing transactions. In this case, the default behavior can be overridden and shut-down proceeds under the control of the service.

More complicated was the adaptation of services to the changed configuration access. Configurations are acquired asynchronously. Existing services had to be adapted to that changed model. Furthermore, code that handles failure situations, where configurations are not available or withdrawn at runtime, had to be added.

8 Conclusion

As more and more services are offered electronically, their management becomes increasingly important. This article identified some unique challenges of managing dynamically cooperating distributed services and their relationships. It described a set of requirements that need to be supported to allow management of those services. Based on these requirements, I developed a software infrastructure that facilitates the creation and the deployment of manageable services. The experiences showed that making dynamic services manageable does not require a great amount of extra work. Compared to its benefits, it is a more than worthwhile activity. I hope that the service management area soon gets the service developers' attention that it deserves.

References

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web Services Web,” *IEEE Internet Computing*, vol. 6, no. 2, Mar. 2002, pp. 86–93.

[2] Sun Microsystems Inc., “Jini Architecture Specification – Version 1.2,” Dec. 2001, `http://www.sun.com/jini/specs/jini1_2.pdf`.

[3] W. Kim, S. Graupner, A. Sahai, D. Lenkov, C. Chudasama, S. Whedbee, Y. Luo, B. Desai, H. Mullings, and P. Wong, “Web E-Speak: Facilitating Web-Based E-Services,” *IEEE MultiMedia*, vol. 9, no. 1, Jan. 2002, pp. 43–55.

[4] M. Garschhammer, R. Hauck, H.-G. Hegering, B. Kempter, M. Langer, M. Nerb, I. Radisic, H. Rölle, and H. Schmidt, “Towards generic Service Management Concepts: A Service Model Based Approach,” in *Proceedings of the 2001 IFIP/IEEE International Symposium on Integrated Network Management (IM 2001)*, (Seattle, WA, USA), May 2001, pp. 719–732.

[5] Telecommunications Information Networking Architecture Consortium (TINA-C), “Service Architecture – Version 5.0,”, June 1997. `http://www.tinac.com/specifications/documents/sa50-main.pdf`.

[6] A. Sahai, V. Machiraju, and K. Wurster, “Monitoring and Controlling Internet based E-Services,” in *Proceedings of the Second IEEE Workshop on Internet Applications (WIAPP 2001)*, (San Jose, CA, USA), July 2001, pp. 41–48.

[7] TeleManagement Forum, “Generic Requirements for Telecommunications Management Building Blocks – Part 1 of the Technology Integration Map (GB909, Part 1), version 3.0,”, Jan. 2001.

[8] D. Lewis, “A Review of Approaches to Developing Service Management Systems,” *Journal of Network and Systems Management*, vol. 8, no. 2, June 2000, pp. 141–156.

[9] P. Hasselmeyer, “Managing Dynamic Service Dependencies,” in *Proceedings of the 12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001)*, (Nancy, France), Oct. 2001, pp. 141–150.

[10] P. Hasselmeyer, “A Novel Architecture for Dynamic Least Cost Routing,” in *Proceedings of the 2000 International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2000)*, (Split, Croatia), Oct. 2000, pp. 379–388.