A Loosely Coupled Federation of Distributed Management Services — Extended Version —

Gerd Aschemann and Peer Hasselmeyer
Darmstadt University of Technology
Department of Computer Science
{aschemann,peer}@informatik.tu-darmstadt.de

ITO TR-00-09

October, 17th 2000

Abstract

This paper describes an architecture for management services. The architecture consists of a number of small components, each performing a highly specialized task. Together, they form a dynamic, highly automated, yet integrated management system. The components are plug-and-play services that use Jini to dynamically establish communication links among themselves and form transient management federations. Our architecture is built around a configuration service which provides for consistent configuration of managed resources. We identify a number of common management scenarios and demonstrate how they can be supported by our system.

Keywords: Event-based management, CORBA-based management, Jini-based management, Java, Web-based management, distributed-systems management, distributed management

1 INTRODUCTION

For many years, management of networks and distributed systems has been shaped on one hand by dedicated management architectures, such as OSI-MF/TMN and SNMP, and on the other hand by monolithic management platforms and frameworks, such as HP Openview, IBM Netview, or CA Unicenter. SNMP's restricted information modeling techniques make designing management applications complex and awkward. Both SNMP and TMN are therefore complex technologies and management platforms supporting one or even both of them are just as complex. Not only do such platforms require appropriate hardware and expensive software, they also need a complex analysis

process, a matching organizational framework and constant customization, configuration, and development. Therefore, only large organizations can afford to invest in this process and the highly specialized human resources to run the systems.

Current and widely used technologies like CORBA, the World Wide Web (WWW), and Java have led to a strong trend of replacing the special-purpose management architectures by multi-purpose middleware infrastructures. These infrastructures allow for cheap off-the-shelf components, such as the CORBA Common Object Services (Naming, Transaction, Life-Cycle, etc.), and are available to a broad range of developers, programmers, and operators. However, we have to distinguish between two basic types of approaches:

- The WWW and related technologies rely on a document oriented information model and a stream oriented communication model. That is, they are basically used to send semi-structured (HTML) information with a simple file transfer protocol (HTTP) to human users (browsers). Several enhancements, e.g., XML, CGI scripts/servlets, and ECMA script/Java applets, enable powerful front end and back end applications. Even though the WWW provides a well-known and widely available user interface, the overall architecture is essentially restricted to a two-tiered client/server approach.

- The other important kind of middleware is represented by CORBA, DCOM, DCE, EJB, etc. These technologies usually provide an object-oriented or at least object-based information model and a synchronous, task-oriented (RPC) communication mechanism. This allows for arbitrary distribution of components and the construction of two-tier, three-tier and even n-tier applications. Objects transparently call well-known methods of other objects based on strong typing and efficient dispatching.

Both approaches have particular strengths and weaknesses. Distributed applications in general, and management applications in particular, often need features of both worlds and strong integration mechanisms are required.

We believe that the general trend towards the use of multi-purpose middleware infrastructures will ultimately lead to an open architecture of small, highly specialized management components. The architecture must allow for the formation of arbitrary management federations and management applications on demand in an automatic fashion, or with only little additional assembly work. It also has to enable easy extensibility by integrating new services as well as legacy management protocols and applications. Administration of resources, services, and management services should be performed mostly automatic to free administrators from tasks better performed by machines.

To investigate the possibilities and challenges of such an architecture, we defined and implemented a set of components which enable a number of example scenarios and represent a base for further management applications. Since configuration management, i.e., deployment and management of management services and relationships among them, is a prerequisite for enabling a management federation, we have grouped our components around a configuration facility.

Due to our incorporation of new as well as existing management services, we did not restrict ourselves to a single communication protocol. Components are currently integrated using CORBA, Java Remote Method Invocation (RMI), and the Hypertext Transfer Protocol (HTTP). User interfaces are built using Java's Swing package as well as the Hypertext Markup Language (HTML) and the Extensible Markup Language (XML). To reach our goal of dynamic, automatic, and ad-hoc formation of management federations we employ the Jini connection technology [24] as our service framework.

The paper has the following structure. Section 2 discusses some specific scenarios where our architecture can be successfully applied, while the following section investigates some requirements resulting from these case studies. We then give a brief introduction to Jini in Section 4. Section 5 presents the design of the architecture, and Section 6 goes into some details of the implementation. Before we conclude in Section 8 we provide a comparison with related work in Section 7. A shortened version of this paper [4] will appear in a special issue of the Journal of Network and Systems Management (JNSM).

2 SCENARIOS

To introduce our architecture, we start by describing a few scenarios which we think are typical network management tasks. By analyzing these scenarios, we can identify the components that are needed to support these tasks. The scenarios are not focused on an administrator's work but describe tasks performed by different people in a medium- to large-sized company. All scenarios are centered around the administration of a work-group's printer. As these scenarios show, human intervention is required only in abnormal cases.

First, let us assume that somebody is printing a large document. After a few pages, the printer runs out of paper. Usually, the person printing the document finds out about the missing paper when trying to pick up the printout. The user has to add some paper and wait some more time for the job to be completed. A better solution is to be notified by the printer as soon as the problem occurs. To support this, the out-of-paper notification has to be dynamically routed to the person that submitted the current print job. There is no need to forward the message to a central administrator as this problem can be better fixed locally.

A similar problem occurring on a regular basis is missing toner. Here, too, the notification should be dynamically routed, in this case to the person in charge of refilling the toner, who is probably not the one who submitted the current print job. Instead, the message could be forwarded to the supply room who can automatically send a new toner cartridge to the appropriate department. If they are out of stock, more toner could automatically be ordered from their supplier. As these two examples show, it is important to send notifications to different destinations depending on their type.

In case of a serious hardware defect in the printer, the fault notification has to be sent to yet another person—the company's system administrator, who can then call a service technician to fix the problem. If the failure occurs while the system administrator is not at her desk, the failure notification should be forwarded to her PDA or cellular phone. Of course, every event should also be logged in a database. The selec-

tion of a notification's destination should usually happen automatically without human intervention.

To find out about the printer's problems, the service technician would connect to the company's network and collect required status information from the printer. This should ideally happen before the technician leaves his office, ensuring that he takes along the required parts. If the problem cannot be fixed on site, a new printer might be installed (either as a temporary or permanent replacement). The initial configuration of such a new printer should be mostly automatic, getting the required information from services in the network. For example, Internet Protocol (IP) addresses can be acquired from a Dynamic Host Configuration Protocol (DHCP) server, printer drivers can be downloaded from the vendor's web site, and users are notified of the new printer by some kind of service broker.

Installing a new printer is also needed if the old one is commonly overloaded. An intelligent infrastructure should constantly monitor the queue size of the printer spooler and notify the administrator if excessive load is regularly detected.

If an administrator discovers that new toner is required every other week, she might want to introduce accounting to monitor usage behavior. In our component-based architecture this would only require starting an accounting service somewhere in the network. The new component would automatically find all printers and ask them to send accounting records to it. Depending on the data format used by the accounting service, it is easy to make the collected data accessible via the Web, so that employees can look at their printing history.

3 ANALYSIS

By analyzing the above use-cases, we can identify a number of properties that our communication infrastructure needs to offer, as well as a number of services that need to be present somewhere in the network.

3.1 Infrastructure Requirements

Our architecture consists of a large number of small components which have to discover and interact with each other at run time. As this is a recurring problem that all components face, the communication infrastructure should support this functionality and ease its usage. For both problems—communication across a network and finding services—a number of solutions exist. Communication can be performed via CORBA, Java RMI, plain sockets, etc. Locating services is possible via COS naming services, RMI registries, and so on. However, these technologies do not completely satisfy our aim. They all require clients and services to know the location (usually an IP address) of the naming service and, as our ultimate goal is to minimize pre-configuration, this is not acceptable. Clients should configure themselves as autonomously and automatically as possible. Jini was found to support this requirement.

3.2 Required Services

From the scenarios described above, we can deduce the components needed to solve the respective management tasks. It is assumed that we are dealing with a "traditional" network printer which is not Jini-enabled, but has other means, like the Simple Network Management Protocol (SNMP), to communicate with the outside world. We consider such a device an "inherited" device, and consequently call mechanisms to access such a device "inherited" protocols or "inherited" interfaces. All other components are assumed to be Jini-enabled. This is of course true for all components that we introduce in our architecture, but currently not for user applications like a word processor.

To integrate a particular inherited device into a network we have introduced the concept of a so-called *Nanny* [3], a virtually unique entity for each device, which takes care of it. A Nanny can be seen as some sort of extended Jini device bay which combines specific knowledge about its protégé with context specific knowledge about the surrounding network. Hence it does not only integrate the inherited device into a Jini framework but into arbitrary networks of services. Since the Nanny is the driving force behind the integration of the device it requires a number of supporting management services according to the inherited interfaces of the device. These are described in the following paragraphs.

SNMP trap service. Our first scenario uses event notifications. To be able to handle them they must first be captured and made available to interested parties. We therefore need a service that accepts SNMP traps, transforms them to Jini events and passes them on to registered listeners. For our printer, its Nanny is such a listener. Since the Nanny knows about the device it cares for, it is able to derive specific information from the generic event object and forward it to appropriate parties, e.g., the printing service. Depending on the type of the event, the receiving service may forward the event to another event listener, e.g., to the user who submitted the print job or the printer administrator. During the integration phase of the printer, the Nanny is responsible for configuring the IP address of the SNMP trap service within the printer.

SNMP gateway service. Not only need SNMP traps be captured, further management via SNMP must be possible, e.g., when a service technician wants to access status information of a printer. An SNMP gateway service must therefore exist which can be used by clients to issue SNMP commands and return the results.

Protocol services. Since the low level and initial configuration of the printer is usually provided through inherited protocols, appropriate Jini-enabled variants of these services must be provided by the architecture, e.g., a DHCP service and a Trivial File Transfer Protocol (TFTP) service. The Jini-enabled variants do not only make them a part of a Jini federation, they also offer appropriate interfaces to allow for configuration and event registration.

Printing service. Because the printer is used by applications, the Nanny should provide a Jini wrapper offering the service of the printer through appropriate interfaces and

attributes. In this case, the printer will be discovered by the already mentioned printing service. However the printing service itself may make use of more generic base services to split up its task, such as a queueing service and a user service to authenticate users and keep track of the issued print jobs.

Accounting service. Enabling accounting requires the start of an accounting service. It collects accounting data from all accounting-enabled services and allows other (authenticated) entities to access the collected data. The accounting service registers with other services when it starts; these services then send it accounting records as necessary. Thus, services supporting accounting must expose methods the accounting service can use to register with them and must send the required data to the accounting service.

Configuration service. Finally, we can foresee that some sort of configuration service is necessary to take care of consistent configuration management. This is described in more detail in section 5.

Further components. Finding out about frequent overloads is a classical monitoring problem. Two parties are involved—one that monitors and one being monitored. Because the monitoring component accesses services but does not offer a service to the Jini federation, it is not registered as a Jini service. Monitored services must expose methods for supplying monitoring data. Depending on the frequency of events, data transfer can follow a push or a pull model. A description of these two models can be found in [14]. If the monitoring component discovered the frequent overload of a printer, it can send an appropriate message, e.g., to the system administrator.

4 THE JINI INFRASTRUCTURE

For a better understanding of the reasons why we chose Jini as our service infrastructure we present an overview of the relevant features of Jini in this section.

Services in a Jini system announce their availability by registering with the system's service repository, the so-called "lookup service" (LUS). The location of lookup services does not need to be known in advance as they can be found at run time by using Jini's multicast discovery protocol. Clients can get references to services by querying lookup services. A Jini federation is formed by all services and clients currently participating in the system. The boundaries of one federation are defined by the reachability of one particular lookup service instance. Throughout this paper we use the terms "service" and "component". There is a slight difference between these two terms. Components are all entities that are part of a Jini management federation, i.e., services as well as clients. Only components that are services in the Jini sense, i.e., entities that register with lookup services, are called services.

Another feature of Jini that we make frequent use of is its protocol independence. Services can use any protocol to communicate with their clients. This functionality is enabled by Java's code migration facilities. Services send protocol engines in form of

proxies to their clients. Clients only need to know the (Java) interface of a service, not the actual implementation of the service or its proxy. This protocol independence allows for the easy integration of inherited services as these only need to be wrapped in a thin Jini layer. Communication protocols do not need to be mediated, converted, or changed.

5 ARCHITECTURE AND MAIN COMPONENTS

From the management point of view a distributed system consists of a set of managed resources which are monitored and controlled by a set of management applications. Due to the very nature of being distributed, each component has its own partial view of the overall configuration, i.e., its own relationships to other components. Hence, this configuration will not necessarily be consistent, that is, arbitrary pairs of components may have different notions of their mutual relationships. This is true even for centrally managed systems or if only one management application exists. We therefore believe that the major challenge for a framework or platform for the management of distributed systems is to preserve the consistency of the configuration, in particular if the management system itself is distributed.

Figure 1 depicts an abstract view of our architecture: The management applications (MA) manage the resources (MR) through a configuration layer which ensures consistent relationships among components. Of course, most management actions (such as performance management or fault detection) do not primarily change the configuration. It is nevertheless useful—at least on the architectural level—to let all events pass through the configuration layer, as most functional management areas use configuration information, e.g., by routing or correlating alarm messages based on the service topology. Consequently, the central or at least virtually central component of our management architecture is the configuration service.

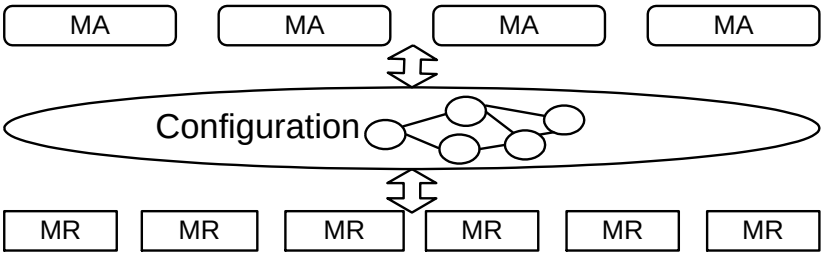

Figure 1: Abstract architecture

5.1 Configuration Service

The configuration service does not only store configuration data but also takes an active part in configuration management. Mechanisms that allow management applications

to change and retrieve (pull) configuration information are therefore required. Furthermore, facilities to actively push information changes and notifications about such changes to interested parties are needed. Pushing information changes to managed resources means to reconfigure them; pushing information changes to management applications makes them aware of configuration changes, such as notifying an administrator about changes in the service topology. To keep the configuration service as simple as possible, we conceptually split it up into several sub-services:

1. A repository which persistently stores configuration data, and provides an interface to change and retrieve data as well as to subscribe to change events;
2. A rule base which contains rules and policies [15] to preserve the consistency of the configuration;
3. A scheduling (time) service which may trigger actions in the repository at certain times, such as to check for subscription expirations;
4. Protocol adaptors to convert configuration information according to the respective information model and/or communication model of the management applications and managed resources.

Due to its implementation history our configuration service is not strictly separated into these parts (cf. section 6.1).

5.2 Observation and Controlling Services

We will not go into the details of all management services. Instead we will try to distinguish between management services that mostly monitor the distributed system and services that change its behavior. Of course, some services act in both roles.

Monitoring. Monitoring is the classical task of most management systems. For the sake of simplicity, monitoring is often performed by management services and management applications through polling the appropriate data from the managed resources (SNMP). Only critical events are forwarded actively by the managed resources. However, only few state changes of the managed resources have a direct impact on the behavior of the system in terms of reconfiguration. Nevertheless, management systems must keep track of the global picture not only to react to emergency situations but also to allow for proactive management to avoid critical situations. Therefore, we enable management services in an observer role to react to changes in a dynamic way according to the needs of the organization running the distributed system. For example, a printer running out of toner may be ignored by the management system in most organizations but may be critical to others, such as in a stock exchange. We therefore strongly encourage the use of events in management to allow for appropriate reactions according to an organization's needs and policies.

Jini helps us provide management-relevant notifications as one of its core features are distributed events. Every service in our system is required to provide notifications and appropriate subscription interfaces. As event sources do not distinguish between different listeners, flexibility in reacting to events in arbitrary ways is ensured.

Control. Event propagation is not only used for monitoring purposes but also to forward configuration state changes to arbitrary targets. Targets are managed resources which may be directly reconfigured, as well as management applications which might take further action, such as performing complex reconfiguration tasks or notifying a human system administrator. State changes might be due to spontaneous events in managed resources or reconfiguration actions by management applications and by human system administrators (see Figure 2). Our management architecture allows management applications to establish arbitrary and complex control loops through our management services, thereby implementing any desired management work flow. Of course, all entities may also offer conventional query and change interfaces.

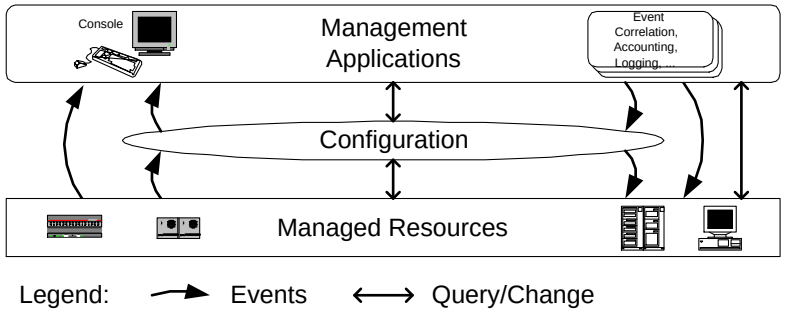

Figure 2: Extended abstract architecture with arbitrary control loops

5.3 Internal Service Architecture and Used Technologies

Figure 3 shows our architecture. We have replaced some of the generic components with their concrete implementations and refer to the technologies used. The figure does not contain all relationships between the management components and the managed resources. Instead, it is restricted to the most important relationships and typical examples.

The various parts, that is, the configuration service, the Nanny sub-architecture, the CORBA/SNMP-gateway, and generic Jini, CORBA and Web services (e.g., the Jini-LUS) are separate entities but are closely cooperating despite being loosely coupled.

As outlined in section 1, internal communication and communication based on distinct method calls to certain objects make strong use of CORBA. If the participating components are native Java objects we also make use of RMI where appropriate, in particular in conjunction with Jini. External representation, desired (Java) code migration, event distribution, and self-configuration rely on appropriate Web technologies:

- Java and HTML (can be replaced and extended by XML) for representation, ...
- HTTP for code migration, stream-based and document-based information structures, file transfer, ...

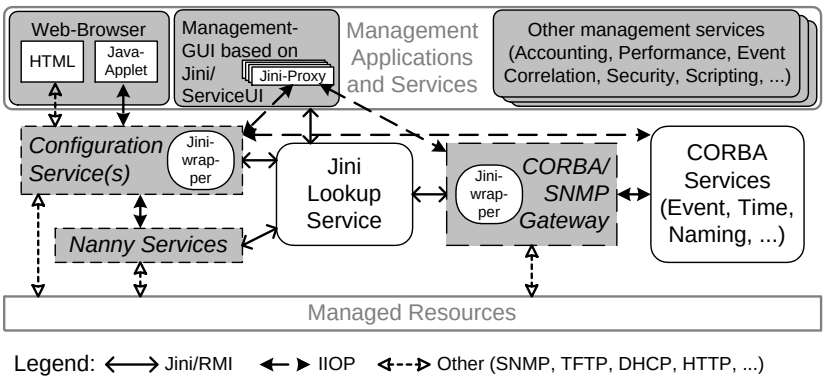

Figure 3: Detailed architecture and most important communication relationships

- RMI for configuration-free integration through Jini, Jini event distribution, communication with Jini-enabled services, ...

6 IMPLEMENTATION

The component-based architecture of our framework was very helpful in implementing the required services. It allowed us to implement individual components separately once the service interfaces had been designed. We were even able to integrate existing components which had been implemented prior to the final definition of the overall architecture. In this section we focus on some details which we consider to be of major interest, in particular the integration of existing components.

6.1 Integration of the Configuration Service

With the System Configuration Tool (SCOT) [5] we had defined and implemented a repository for configuration information. SCOT provides both a Web interface for human users and a CORBA interface to access configuration data by management applications [7] and enhanced user interfaces, i.e., Java applets.

SCOT has been integrated with the Jini-based architecture (cf. Figure 4), in particular

- the repository reacts to external changes, that is state changes in managed resources and/or management applications, and
- the repository sends events if the state of an object in the configuration repository is changed, such as through external events or regular changes.

We realized this by adding two optional functions to each object in the repository, which are empty by default and are therefore not evaluated. Since the repository is

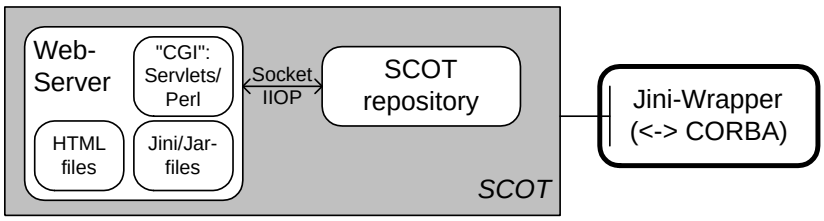

Figure 4: SCOT integration into Jini-based federation

implemented in Lisp these functions can actually be arbitrary Lisp operations. One of the functions is called when an external event occurs. It implements a push consumer interface of the CORBA Event Service specification [16] and gets the event data as a `CORBA.Any` object. If the state of the object is changed the other function is automatically called with the address of a CORBA push consumer as parameter, i.e., an associated CORBA event channel (see below). This function should call the consumer with an appropriate `CORBA.Any` object as a parameter that contains information about the state change. The repository allows to store arbitrary configuration data by means of extended Lisp objects. Therefore, it is not possible to provide more specific, i.e., stronger typed, interfaces for event data in a generic way. Further event handling, i.e., queueing of events, is performed through two standard CORBA event channels associated with each object, one channel for incoming event messages and one channel for outgoing events. The integration with Jini is done outside of the configuration service, within the wrapper. The wrapper is an event consumer for each repository object. Interested parties can register with the wrapper for state change events and the wrapper performs lease [26] handling, multiplexing, and delivery of the events as Jini events.

As mentioned in Section 5.1, our SCOT repository contains the configuration information together with associated rules (consistency rules and policies). Thus, it does not separate sub-services, but allows for the encapsulation of data and code as is usual in object-oriented systems.

6.2 SNMP Service

We already had a CORBA/SNMP gateway [7] that works similar to the Joint Inter Domain Management (JIDM) approach [27]. It is implemented in C++ and allows to access SNMP-enabled devices through CORBA and to forward SNMP traps and notifications as CORBA events. As one can imagine, such a gateway is much slower than direct SNMP messaging. For simple variable access direct SNMP is around 40 times faster than through the gateway, for tables this factor is reduced to 4. [7] contains exact numbers.

The integration into our component architecture adds a wrapper component to the existing gateway components (see Figure 5). The wrapper has two tasks:

SNMP gateway. As a “proper” Jini service, the Jini wrapper joins the Jini management federation by registering with Jini lookup services. Clients can find the

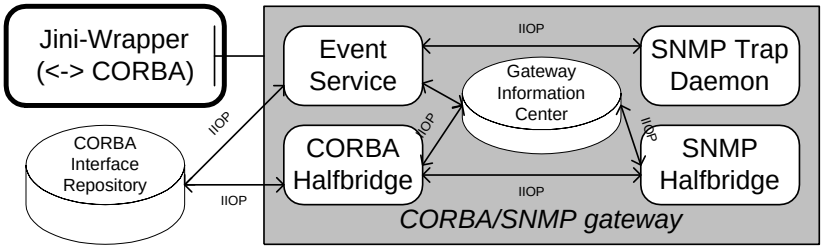

Figure 5: Integration of the SNMP Service

CORBA/SNMP gateway via a Jini lookup service and register for events or access other functions of the gateway, such as getting some information from an SNMP device.

SNMP trap service. The CORBA/SNMP gateway accepts SNMP traps and notifications, transforms them into CORBA objects and sends them to a CORBA event channel. The CORBA-to-Jini wrapper registers with this event channel and relays the event objects to all interested Jini listeners (see Figure 6). The event objects sent to listeners contain all relevant data of incoming SNMP traps and notifications. Due to limitations of the CORBA/SNMP-gateway implementation the encapsulated notifications are actually plain CORBA data structures (`CORBA.Any`-objects). They are mapped to a generic `Trap` class, determined by the translation specification of the CORBA/SNMP-gateway. This could be extended by mapping the SNMP notifications to specific CORBA types and making use of CORBA typed event channels.

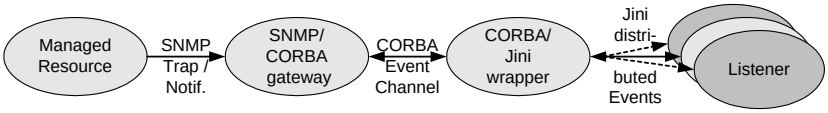

Figure 6: SNMP to CORBA to Jini Event flow

6.3 The Nanny sub-architecture

Our already mentioned Nanny architecture [3] is a small federation of Jini services enabling the integration of inherited devices, e.g., printers, X-terminals, laptops and other mobile devices into an environment of other services [6], and for management purposes (cf. Figure 6.3). Since it was based on Jini services from the beginning it could easily participate in our overall architecture.

The CORBA/SNMP gateway encapsulates management of devices through SNMP

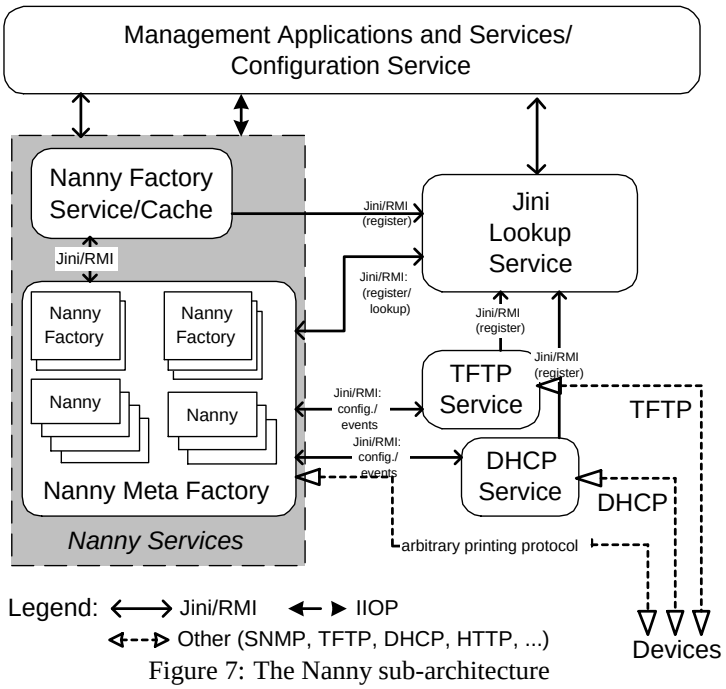

Figure 7: The Nanny sub-architecture

as a Jini service. A Nanny that takes care of a device managed through SNMP integrates the device into the management environment in various ways, e.g.:

- The Nannies announce their service as an event supplier through the Jini LUS. Interested parties, such as a management GUI or an event correlation service, register with the Nanny as event consumer. The “Hen-and-Egg”-problem is solved by the Jini LUS in a smart way. If the Nanny has already registered with the LUS an interested consumer detects it by looking up services with an event supplier interface. Additionally the consumer registers with the LUS as being interested in services providing an event supplier interface. If such a service registers with the LUS in the future the consumer is informed by the LUS about the new supplier and can register with it for events. Hence, to bring the services together, it finally does not matter who comes first.

- A performance management service may frequently request appropriate data from the available devices. It therefore asks the LUS for a list of Jini services which provide such information. Although represented as Jini services, performance data always originates from a physical device like a PC or a router. Some of these devices will be Jini-enabled and provide the required information directly. Others will be inherited devices which are not Jini-enabled but can be managed via SNMP by a Nanny. The Nanny will perform the request on behalf of the inherited device by calling the SNMP/CORBA gateway. The SNMP/CORBA gateway finally forwards the request to the real device as an

SNMP request. The resulting SNMP reply is received by the gateway and sent back to the Nanny as CORBA result message. From the viewpoint of the performance management tool there is no difference between a native Jini-enabled device and an inherited device.

Some of the contained services do not only serve as part of this sub-architecture, i.e., through the Nannies, but are directly used by other parts of the management federation. The configuration service and some management GUIs, for example, directly register with the DHCP and TFTP sensors to detect new devices, even if there is no Nanny finally taking over control.

6.4 Revisiting the Sample Scenarios

Coming back to the scenarios of Section 2, we show how typical management problems are solved by our management federation.

6.4.1 Forwarding temporary problems to the appropriate person

During the initialization phase of a network printer an appropriate Nanny has accepted the responsibility for the device. The Nanny retrieves required configuration information for the unit itself and its environment, e.g., the closest printing service and SNMP trap daemon of a CORBA/SNMP gateway. It registers the printer with the print server as new worker and forwards some of the configuration information to the device in an appropriate manner, e.g., by DHCP or TFTP. It then registers for event notifications with the Jini backend belonging to the CORBA/SNMP gateway. In case the printer sends an SNMP trap about an exceptional situation, the Nanny is able to interpret the trap and to construct the appropriate context. For example, if the printer is running out of paper, the Nanny can ask the printing service about the owner of the current job and forward the message to her. If the problem were missing toner, the Nanny would ask the configuration service for the administrator on duty and propagate the event to that person.

6.4.2 Helping to solve serious problems

If a more serious problem occurs, e.g., if some mechanical part of the printer breaks, additional action may be actively aided by the management environment. In the moment it becomes clear that there is a problem which cannot be solved locally, all information about the problem is entered into a trouble ticketing system—at least a problem description and a unique identifier for the broken device like its medium access control (MAC) address. Through the unique identifier additional information can be associated with the trouble ticket, e.g.,

- the type of device,
- the responsible hardware support company,
- the location of the device,

- the responsible local system administrator.

Opening the trouble ticket usually initiates further action:

- The device is disabled in the configuration repository. This action triggers reconfiguration of the printing facilities in the workgroup containing the broken printer. Current print jobs are rerouted and future print jobs will be directly sent to another printer—maybe belonging to a neighboring workgroup. Users are notified by e-mail that they have to fetch their printouts at another location.

- The problem is forwarded to the support company which assigns a technician to it. The technician gains further information about the device through an management access point which is open for external access according to the security policies of the client company. It could, for example, allow to request the number of printed pages and other state information directly from the printer through the Nanny and the other components as outlined in Section 6.3. When the technician visits the company he brings the necessary parts with him to repair the printer. If the printer cannot be fixed locally he has a replacement unit with him and has provided the necessary information for the integration of the replacement device to the responsible system administrator beforehand. If that person has entered the information into the configuration repository, the technician can simply plug the new device into the network, power it up, and the workgroup can almost instantly use the new printer. Reconfiguration of the printing facilities to use the replaced printer is triggered by the detection of the device through its Nanny.

7 RELATED WORK

Java-based management. Over the last years, Sun Microsystems has started several efforts to make Java an enabling technology for distributed systems management. The first one, the Java Management API (JMAPI, [17]) was suddenly given up in 1998 without any obvious reason known to us. However, it contains some interesting approaches, in particular a hierarchical event model, where subscribers can specify in various degrees which events they want to see. Sun took up some of the ideas of JMAPI in the newer approaches and dropped others. The newer approaches are the Java Dynamic Management Kit (JDMK, [22]) and the Java Management Extensions (JMX, [23]). Although both have been developed independently in the beginning, the latter is considered an abstract architecture for management in general while the former is seen as an implementation of this architecture, at least on the agent and instrumentation levels.

JDMK provides a framework for composing management agents from generic and specific management components using Java Beans [18], Java's native component technology. It enables deployment of management components within the agent at runtime, thereby migrating part of the management intelligence to the agent. The framework abstracts from particular communication protocols and allows agents to communicate with its manager through arbitrary protocol adapters (SNMP, HTTP, RMI, ...). It furthermore allows for the cascading of agents which serve as managers to other agents.

While JDMK is a product which can actually be purchased, JMX is an abstract architecture, which has a reference implementation but is intended to have multiple implementations by different vendors. It splits up management into three different levels: a) an instrumentation level, b) an agent level, and c) a manager level. On a) we find management instrumentation, i.e., management interfaces of arbitrary Java objects. Management of non-Java resources can be achieved via the Java Native Interface (JNI, [20]). Instrumentation is encapsulated by so-called management beans (MBeans). On the agent level (b), JMX proposes MBean servers that are container objects providing a certain runtime environment for MBeans. MBean servers use protocol adapters to allow manipulation of MBeans by various manager components, e.g., Web browsers, and via proprietary and standardized management protocols, such as SNMP, CMIP/TMN, CIM/WBEM, etc. In addition to MBeans used for instrumentation, MBeans can be dynamically loaded from a manager to perform certain management actions inside the agent. The manager level (c) is not yet specified.

Compared to our approach, JMX and JDMK are more complementary than competitive technologies. They currently support the implemention of agents while our architecture tends towards the manager side. In particular, we do not address the implementation of agents for Java-enabled devices. JMX/JDMK, on the other hand, do not directly address configuring devices on a low level and making them part of a Jini community. Both approaches deal with non-Java-enabled inherited devices, though. Here, our goal is to automatically configure these devices and integrate them into Jini federations. JMX would only allow these devices to be administered via a number of protocols. Additional management software, e.g., our Nanny infrastructure, would be needed to configure the devices and make them Jini citizens. Since we are open to existing and new management protocols, it should be possible to integrate JMX-based managed resources just like other management agents.

On the architectural level JMX and our architecture have a number of significant differences. Our use of Jini allows for arbitrary and dynamic distribution of components. MBeans are more restricted as they can only easily access MBeans that reside in the same MBean server. Interaction between MBeans of different MBean servers is not spontaneous and needs to be configured. The question how managers find agents to be administered is not addressed by JMX. Jini solves this problem inherently. In the same vein, JMX only supports local event propagation. Jini offers distributed events which allow for more freedom in component distribution.

Jini-based management. Sun has shown that Jini and JDMK can be used together to manage Jini-enabled services and devices through JDMK agents [19]. Manageable Jini-enabled entities register their proxy objects, which implement a management interface, with the Jini lookup service. These proxies are discovered and downloaded by the JDMK-based management agent. Jini devices or services can now be managed through the management protocols enabled by the management agent. Thus, the device or service is a management-instrumented Java resource to the agent. This work is aimed at the integration of Jini-enabled entities into management, while our approach covers the integration of inherited devices. Furthermore, it is important to note that there is a conceptual difference to our architecture. We do management *with* Jini, not

of Jini.

Additionally, Sun proposes a new architecture on the manager level (according to the JMX terminology), the so-called "Federated Management Architecture" (FMA, [21]). As of 2000, there are no practical experiences known to the authors, although a reference implementation became available recently. Despite being specifically aimed at distributed management, FMA actually only introduces a number of Jini-based but general distributed-systems services, e.g., security services, persistency, concurrency control, logging, etc., which are well-known from other middleware architectures.

As a main feature, FMA introduces so-called "stations" that host management services and offer them a well-known operating environment. Stations are therefore analogous to Enterprise JavaBeans (EJB) containers. In addition to hosting services, stations play a key role in FMA's fault-tolerance, security, concurrency control, and remote instantiation mechanisms. All these mechanisms require communication to be mediated by the station hosting the target service. This is achieved by adding an appropriate layer on top of RMI. RMI is therefore the only usable protocol in an FMA system. Emphasizing FMA being a general middleware platform, an information model—which is generally considered important for management architectures—is not specified. The other Java-based management architectures (JDMK/JMX) are based on Java-centric information models and could be used in conjunction with FMA.

Similar to our approach, FMA proposes a management framework where the management itself is distributed. Some of the services introduced by FMA, e.g., logging and deployment, are not yet existent in our system and would enhance it. Most of the proposed mechanisms do not fit into our model, though, because they are restricted to the augmented version of RMI. One of the main advantages of Jini (and one of the main reasons for us using it)—protocol independence—is lost in FMA systems. Incorporating inherited components in an FMA system is therefore more complex than in a plain Jini system. Looking at the event model, we find that FMA uses the Jini/RMI communication model and only extends it by an event multiplexer. We demonstrated a similar solution by the integration of a CORBA event channel. In contrast to FMA, our architecture provides an information model [5] which is largely based on the requirements of configuration management but which is also usable in other functional areas of distributed-systems management through its integration with CORBA and the mapping to HTML or XML.

However, it is important to note that FMA is backed by some vendors in the field of Storage Area Networks (SAN), the Jiro initiative [12].

Component architectures. Lewis et al [13] describe a management system comprised of a set of interacting components. The system introduces plug-and-play functionality by the use of a so-called "integrator". This component connects various components at runtime. Our architecture employs Jini to achieve that functionality. Their system uses the event model of the CORBA component model which follows a publish/subscribe/distribute paradigm just as the Jini distributed events used in our architecture. An additional advantage of using Jini distributed events is the handling of partial failure. All event registrations are bound to a lease and have to be renewed regularly by the listener. If a listener becomes unreachable and does not renew its lease,

it is automatically discarded as soon as the lease expires. Crashed listeners do therefore not consume resources unnecessarily over an extended period of time. CORBA events do not have this feature.

Feridun et al [8] describe an approach to distributed management using lightweight mobile components that can run anywhere in the managed system. Components are run on participating nodes in dedicated environments and are located via a distributed directory. Most of the infrastructure functionality is provided by the Voyager middleware. It is therefore comparable to our use of Jini, but it offers a slightly different set of services, e.g., Jini incorporates events for changes in the set of available components but does not handle persistency or migration. Their use of a dedicated environment allows for lifecycle management, i.e., installation, migration, and removal, of components. A similar scheme could easily be integrated into our architecture and would enhance it. Our use of a configuration service helps keeping the managed system in a valid state. This is not addressed in their architecture.

Our Nanny architecture is an extension to the device bay [25] concept of Jini. The Jini Surrogate Project [11] also defines an extended device bay architecture. In particular, it specifies an additional lightweight protocol, the Jini Technology IP Interconnect protocol [10] based on UDP to integrate small devices which are not able to incorporate a native Java virtual machine. This is the main difference to our Nanny architecture, since we try to integrate inherited devices with their inherited protocols (DHCP, TFTP, SNMP) instead of defining a new protocol to integrate them into a Jini federation.

Web-based approaches. Marvel [2, 1] is an architecture for managing resources at the service level rather than at the element level. It aggregates information retrieved via low-level protocols into higher level objects. It uses Java and its dynamic code loading functionalities for extending the system while operating. Although it allows arbitrary clients to access aggregated management information, Marvel is mainly geared towards human access via the World Wide Web using HTML and Java applets. In comparison, our architecture tends more towards programmed service interaction but allows WWW access via specialized display components. Marvel mainly addresses supervision while our approach deals with configuration management. Marvel, too, uses a publish/subscribe/distribute paradigm for event propagation.

Other automatic service location facilities. The Service Location Protocol (SLP [29]) allows finding the location of arbitrary services within an IP network. Its bootstrapping and trading facilities can be compared to Jini but it only provides information in the form of name-value lists instead of objects. Universal Plug and Play (UPnP [28]) is a recently announced service trading infrastructure built on top of HTTP-based multicast-protocols. Services register their Uniform Resource Locators (URLs) with a central Simple Service Discovery Server together with a type description standardized by the Internet Engineering Task Force (IETF). Clients query this server to obtain URLs of the desired type. Neither UPnP nor SLP have mobile proxies (for protocol independence and easy integration of legacy systems) and distributed events (for monitoring). For our work, they are therefore no viable alternatives to Jini.

8 CONCLUSION

In this paper we described our architecture of small distributed management services. We showed how it supports various network management scenarios and how it compares to other management architectures and frameworks.

It should not be neglected that there is a downside to the spontaneity introduced by our architecture. Everybody who has access to a computer connected to the network can start and access management services. While this is desired in many circumstances, it opens the door for misuse as well. Network management can be easily and effectively disabled by introducing a fake service, e.g., a configuration service that supplies incorrect data. This problem could be solved by the use of authentication. Most management components do not keep separate accounts for individual administrators. It is usually enough to prove the possession of administrator's rights, e.g. via passwords or keys. It therefore appears to be adequately secure for most purposes to require every component to identify itself to the infrastructure, i.e., the Jini lookup service, and let the infrastructure enforce access restrictions. We described such a solution in [9] but its applicability has to be further investigated.

Figure 8: Snapshot of the Management Console

Another challenge for our architecture is scalability. With a growing number of services there is the risk that the overview of service location and component interaction is lost. Means for adequate management of Jini services especially addressing the mentioned problems have to be investigated. We partly addressed the problem of unknown service location by adding appropriate management data to services and by allowing a generic Web-enabled management console to display that data. Service-specific administration can be performed by displaying graphical user interfaces attached to services. Figure 8 shows a snapshot of this console displaying some of our management

services.

ACKNOWLEDGMENTS

We would like to thank Ron Bourret, Alejandro Buchmann, Roger Kehr, Friedemann Mattern, and Andreas Zeidler for proof-reading and discussion of earlier drafts of this paper. We would also like to thank the reviewers and the editors of the JNSM special issue in which a shortened version of this article will appear [4] for their valuable help.

References

[1] Nikolaos Anerousis. An Information Model for Generating Computed Views of Management Information. In *Proceedings of Ninth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'98)*, pages 169–180, October 1998.

[2] Nikolaos Anerousis. Scalable Management Services Using Java and the World Wide Web. In *Proceedings of Ninth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'98)*, pages 79–90, October 1998.

[3] 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 *Proceedings of Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, volume 1700 of *Lecture Notes in Computer Science (LNCS)*, pages 257–268, Heidelberg, October 1999. Springer-Verlag.

[4] Gerd Aschemann and Peer Hasselmeyer. A Loosely Coupled Federation of Distributed Management Services. *Journal of Network and Systems Management (JNSM)*, 9(1):51–65, March 2001.

[5] Gerd Aschemann and Roger Kehr. Towards a Requirements-based Information Model for Configuration Management. In *Proceedings of 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 181–189. IEEE Computer Society Press, May 1998.

[6] Gerd Aschemann, Roger Kehr, and Andreas Zeidler. A Jini-based Gateway Architecture for Mobile Devices. In Clemens H. Cap, editor, *Proceedings of Java-Informationstage (JIT'99)*, Informatik aktuell, pages 203 – 212, Heidelberg, September 1999. Springer-Verlag.

[7] Gerd Aschemann, Thomas Mohr, and Mechthild Ruppert. Integration of SNMP into a CORBA- and Web-Based Management Environment. In *Proceedings of Kommunikation in Verteilten Systemen*, pages 210–221, Heidelberg, February 1999. Springer-Verlag.

[8] M. Feridun, W. Kasteleijn, and J. Krause. Distributed Management with Mobile Components. In *Proceedings of Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, pages 857–870, May 1999.

[9] Peer Hasselmeyer, Roger Kehr, and Marco Voß. Trade-offs in a Secure Jini Service Architecture. In *Trends towards a Universal Service Market (USM 2000)*, volume 1890 of *Lecture Notes in Computer Science (LNCS)*, pages 190–201, Heidelberg, September 2000. Springer-Verlag.

[10] Jini Technology IP Interconnect Specification. `http://developer.jini.org/exchange/projects/surrogate/IPInterconnect.pdf`.

[11] Project Surrogate (Jini). `http://developer.jini.org/exchange/projects/surrogate/`.

[12] Jiro Technology. `http://www.jiro.com/`.

[13] David Lewis, Chris Malbon, George Pavlou, Costas Stathopoulos, and Enric Jaen Villoldo. Integrating Service and Network Management Components for Service Fulfilment. In *Proceedings of Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, volume 1700 of *Lecture Notes in Computer Science (LNCS)*, pages 49–62, Heidelberg, October 1999. Springer-Verlag.

[14] Jean-Philippe Martin-Flatin. Push vs. Pull in Web-Based Network Management. In *Proceedings of Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, May 1999.

[15] Jonathan D. Moffet. Specification of Management Policies and Discretionary Access Control. In Morris Sloman, editor, *Network and Distributed Systems Management*, pages 455–480. Addison-Wesley Publishing Company, 1994.

[16] Object Management Group, Inc. (OMG). *CORBA Event Service Specification*. December 1997.

[17] Sun Microsystems Inc. Java Management API. `http://java.sun.com/products/JavaManagement/index.html`.

[18] Sun Microsystems Inc. JavaBeans. `http://java.sun.com/beans/`.

[19] Sun Microsystems Inc. Jini Technology and the Java Dynamic Management Kit Demonstration. `http://www.sun.com/software/java-dynamic/wp-jdmk.kit/`.

[20] Sun Microsystems Inc. Java Native Interface Specification. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html`, May 1997.

[21] Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA. *Federated Management Architecture (FMA) – Version 1.0 – Revision 0.4*, January 2000.

[22] Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA. *Java Dynamic Management Kit White Paper*, April 2000.

[23] Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA. *Java Management Extensions Instrumentation and Agent Specification, v1.0*, July 2000.

[24] Sun Microsystems Inc. *Jini Architecture Specification – Version 1.1*. `http://www.sun.com/jini/specs/jini1_1.pdf`, October 2000.

[25] Sun Microsystems Inc. *Jini Device Architecture Specification – Version 1.1*. `http://www.sun.com/jini/specs/devicearch1_1.pdf`, October 2000.

[26] Sun Microsystems Inc. *Jini Technology Core Platform Specification – Version 1.1*. `http://www.sun.com/jini/specs/core1_1.pdf`, October 2000.

[27] The Open Group. *Inter-Domain Management: Specification Translation & Interaction Translation*, January 2000.

[28] Universal Plug and Play Homepage. `http://www.upnp.org/`, 1999.

[29] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol (SLP)*. Internet RFC 2165, June 1997.

BIOGRAPHIES

Gerd Aschemann received his Diploma degree in Computer Science at the Darmstadt University of Technology in 1995. From 1995 to 2000 he has been working as a research assistant in the Distributed Systems Research Group of the same University. He is writing a PhD thesis on configuration management in distributed systems. Currently he is starting a new career as an independent consultant in systems management and Internet technologies. His main interests besides network management and distributed systems management are open middleware infrastructures.

Peer Hasselmeyer received a Master of Science degree in Computer Science from the University of Colorado, Boulder, in 1995. In 1997, he graduated from Darmstadt University of Technology, Germany, and received a diploma degree in Computer Science. Since then, he has been a research assistant and PhD candidate in the Computer Science department of that University. His research interests include spontaneous networking, middleware, and network and service management systems.