A Framework for the Integration of Legacy Devices into a Jini Management Federation

Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer,
Roger Kehr, and Andreas Zeidler

Darmstadt University of Technology
Department of Computer Science
{aschemann,domnitcheva,peer,kehr,az}@informatik.tu-darmstadt.de

Abstract. The administration of heterogeneous networks with many devices is a tedious and time-consuming task. Today's approaches only provide static configuration files and make the addition and removal of devices a manual chore. In this paper we present a framework for the integration of legacy devices based on *Jini*, Sun Microsystem's new technology for federating network devices and services. We introduce extended proxy objects called *nannies* that take care of non-Jini-enabled devices and handle the relevant management events, guide a device through bootstrapping, register it with the lookup service, and provide the implementation of the administrative interfaces of the Jini API. Through this approach both Jini-enabled and legacy devices can be handled homogeneously in a *Jini Management Federation*.

1 Introduction

The management of networks and distributed systems is a difficult task that is complicated by the heterogeneity and huge number of legacy devices and protocols in use today. Standardized management architectures, such as the Internet management, i.e., SNMP [3], or the ISO/OSI management model [13] are steps in the right direction, but do not provide for flexible integration and removal of typical devices, e.g., printers, X-terminals, or laptops in a LAN environment. In particular, initial bootstrapping of devices, management of IP addresses, and use, administration, and shut-down of a new device are not handled in an integrated and flexible manner.

We illustrate this point with the daily life-cycle of a typical network printer. For normal operation of the printer an IP address must be assigned and it must be configured appropriately. The IP address is either known by the printer, e.g., through manual configuration, or it can be assigned dynamically, e.g., by DHCP [6] or BOOTP [9] from a pool of IP addresses. An appropriate server assigns an IP address, a host name, and some other parameters to the booting device. Additionally it may refer to other services, e.g., TFTP [14], where the device will find further files for download, like fonts, profiles, or kernels. Printers are usually assigned to a spool service for queueing print jobs, which also notifies the users about successfully finished print jobs. The management system obtains information about the device status through a management protocol, e.g., SNMP, and provides it to management applications or for visualization purposes.

Theoretically, a printer reports to the DHCP service to release its IP address at shutdown. In practice, the device is often turned off without disposing of the address, thus keeping the address allocated until the DHCP lease times out.

Taking the DHCP server as an example, we illustrate the drawbacks of this approach in general. The server tries to perform several tasks: it implements the protocol engine, dynamically binds IP addresses and optionally other parameters to network devices by a hard-coded policy, and guarantees durability of these bindings. Generally it relies on a priori knowledge of the specific devices to be configured and is only flexible to some degree in the address it finally assigns. The configuration is manually edited and can contain references to additional services, e.g., TFTP or SLP [20], but does not care whether these services are set up or properly configured. Finally, the services are usually not embedded in an appropriate management environment, such as an SNMP-enabled management platform. Due to this lack of integration, manual intervention is often required by the system administrator.

Jini is a recently released network infrastructure from Sun Microsystems that enables arbitrary federations of distributed services. Thus, it can be utilized to allow for better integration of tailored management components. In particular, Jini is quite appealing for bootstrapping and integrated configuration management of devices. However, Jini is primarily conceived to function with Jini-enabled devices and ignores the vast pool of legacy devices currently in use. In this paper we present a framework that makes it possible to bring non-Jini-enabled devices into the Jini world.

In Sect. 2 we give a short overview of Jini. Section 3 presents a vision of how non-Jini devices can be integrated. In particular, we address problems of bootstrapping and configuration management. Section 4 introduces the proposed architecture while Sect. 5 illustrates its use with a concrete example. Section 6 discusses related work and Sect. 7 presents our conclusions and an outline of future work.

2 Jini

Jini is a recently released network infrastructure from Sun Microsystems and seems to be very appealing – if used appropriately – for the task of bootstrapping and managing devices in an integrated manner.

Bootstrapping. Part of the bootstrap-phase of a native Jini-device is straightforward and a "built-in"-feature of all Jini-devices: Jini-enabled devices or services announce themselves in the local *federation* of services. Devices find and register with lookup services [19] using the *Jini Discovery and Join Protocol* [18]. Running these protocols currently requires a device to implement at least a TCP/IP stack, a Java Virtual Machine (JVM), and it must be initially configured with an IP address.

Bootstrapping a non-Jini-device is more complicated and discussed in greater detail throughout the following sections. In this section we want to discuss the Jini building blocks that contribute to management as a whole and give a hint why we put great effort into *masquerading* legacy devices as some kind of native Jini-devices.

Jini and Management. From a management perspective the lookup service is the central starting point for management activities. The lookup service interface allows Java listener objects to register for notifications about newly available services. The registration and notification follows the *observer* design pattern [8].

New services that have not yet been configured announce themselves in the so-called *public* group. Administrators then configure these new services according to their local policy.

Basically, configuring a Jini device consists of configuring information relevant to the Jini federation: groups the device should be subscribed to, entries the device is registered with in the lookup service, the device's service identifier, and so on. This is part of the federation management in Jini.

Essentially, Jini contributes to management with the following four features:

- **Lookup Service.** The lookup service acts as a central repository for services. Appropriate means are offered to query and select services based on interfaces, entry types, and entry values. Some aspects of the lookup service are comparable to the CORBA Trading Object Service [12].
- **Proxy Objects.** Services upload serialized Java objects called proxies to the lookup service. These objects can be downloaded to any JVM and invoked to access the service. The proxy acts as a mediator to the service itself, and may implement programmatic interfaces as well as graphical user front-ends for the service. The proxy encapsulates any protocol used for the actual communication between the proxy object and the service.
- **Notification Mechanism.** All services are required to register with the lookup service and – at least in principle – any Java object can register to be notified about changes in the set of services within a lookup service. This makes it possible to implement almost any mechanism to perform management activities on top of Jini.
- **Leases.** Leases are time-based contracts between two objects within Jini. A lease grantor can bind a service to a lease holder for a certain amount of time. The lease holder can use the granted service within this period according to the contract made, but has to renew the interest for the service granted, i.e., renew the lease before it expires. Failing to do so automatically cancels the contract. As any Jini service has to lease its entry in the lookup service, the lease renewal can be used as a *heartbeat*-mechanism and contributes to the robustness of the distributed system as a whole.

3 A Vision of a Jini-Enabled Management Federation

One of our criticisms of traditional management systems is their monolithic and inflexible nature. In contrast to this, Jini offers an infrastructure that makes it possible to restructure management services as aggregations of special purpose components. Thus, it allows for a complete separation of concerns. We use this approach to split up the management services into small components, each tailored for a single purpose. In our approach, DHCP is reduced to a protocol engine. Assigning addresses and other parameters is left to a configuration service.

With Jini such discovery and embedding is available for Jini-enabled devices, even though a dynamic low level configuration (at least the assignment of an IP address) is beyond the scope of Jini. We show how the appropriate properties of Jini can be utilized to provide even such low level configuration and to better integrate non-Jini devices. Therefore, we introduce the term *Jini management federation* by extending the term *Jini service federation*.

A Jini management federation is a loosely coupled group of services which enable distributed administration of other managed services. These services need not necessarily be Jini services. Jini management services could be almost any services which are normally provided by management platforms, such as databases and repositories (inventory, network topology, configuration, etc.), protocol engines (SNMP, CMIP [4], etc.), monitoring tools, or generic user interfaces. As with all other Jini services, the management services need not be native Java services or Jini services but only be encapsulated by appropriate proxy objects. For the main scope of this paper – bootstrapping and initial configuration – the following services are particularly important:

- **Configuration Service.** A configuration service would provide configuration information to other services, by means of a standardized, but extensible interface. This allows for arbitrary implementations, e.g., a more or less static configuration within the boundaries of DHCP as well as highly dynamic configurations that are based on knowledge of local policies, network topology, current resource utilization, etc. With SCOT [1] we have proposed such a repository which can be easily integrated into the Jini management federation.
- **SNMP Gateway.** A generic gateway for SNMP may translate remote method calls (Java RMI or CORBA based) into SNMP requests and return the SNMP replies. SNMP traps may be converted to Jini events which may be forwarded to appropriate event listeners according to a local policy, e.g., a generic Jini enabled messaging service to present them to a user. Our CORBA SNMP gateway [2] is an example of such a generic gateway. Since Jini is not restricted to native Java communication methods like RMI but easily integrates with CORBA systems, a Jini encapsulation of arbitrary CORBA services can be provided without much effort.
- **Persistence Service.** To ensure that the system works properly, most services have to keep (at least parts of) their state persistent. A DHCP service is obviously in trouble if it loses the information about which IP addresses have already been assigned. One solution is the delegation to a persistence service that offers storage and retrieval facilities for arbitrary configuration data. Thus, a Jini-enabled DHCP service could further contribute to the separation of concerns by leaving it to the persistence service to make the final assignment persistent. For reasons of reliability the persistence service could be implemented in a redundant way. This makes administration of configuration data much easier as the data can always be found at a well-known place.

Based on this infrastructure, new network devices should be configured by appropriate services instantly and automatically after they are plugged into the network. Additionally, the management federation should enforce adding notification interfaces to the vast majority of services. This enables arbitrary objects to receive notifications about interesting events.

4 Proposed Architecture for Integrating Legacy Devices

In this section we describe how to integrate legacy devices into a Jini management federation by means of infrastructural components, their interaction, and communication patterns.

Many existing devices do not meet the requirements for a native Jini service, most obviously due to the fact that few devices implement a full JVM. The *Jini Device Architecture Specification* [17] discusses in some detail how such devices can be brought into a Jini environment.

- **Device Bays.** One proposal is a so-called *device bay* that implements a physically co-located JVM to which several devices can be connected. Proxy objects are uploaded from the device to the JVM to implement the protocol between the device bay and the device. The device bay then acts on behalf of the device to run the relevant discovery and join protocols including the subscription to the lookup service.
- **Network Device Bays.** Another option is to physically separate the device bay from the devices. In this case the device bay is running somewhere in the network and devices must initially run some protocol in order to get in touch with the device bay. After some negotiation between the device and its bay a proxy object could again be uploaded implementing the device protocol.

Most legacy devices we have in mind are already network-enabled. A physically co-located device bay could be implemented as a hardware device placed between the printer and the network. Such a solution might be applicable for certain devices but we think that a pure software solution is less costly and more flexible in general. Therefore, we are more interested in providing an architecture for a software network device bay that provides the same features as a hardware solution. It should be possible that newly attached legacy devices are automatically detected and bound to some kind of network device bay. Appropriate management activities could then be started to configure the device and to integrate it into a Jini federation.

What essentially is missing is the initial *event* that activates an automatic configuration process for legacy devices. Some kind of a *sensor* that detects new devices as they are attached to a network or turned on manually could supply these events. We believe that services such as DHCP and TFTP can provide this initial event if appropriately equipped with instruments. We therefore propose an architecture that supports the integration of legacy devices into a Jini environment by means of *sensors* and *binders*. An overview of the general architecture is depicted in Fig. 1.

Binders and Sensors. Binders are services that provide initial bootstrapping information to devices. The most classical services of this kind are DHCP, TFTP, RARP [7], SLP, and others. These are the services that are contacted by new devices first. Equipping these services with instruments results in the construction of sensors that notify interested parties about relevant management events. Usually, binders act upon concrete events related to the protocol they implement. For example, a DHCP server understands several different messages from the device requesting configuration information. If the device accepts the configuration information offered by the DHCP service an appropriate management event can be supplied.

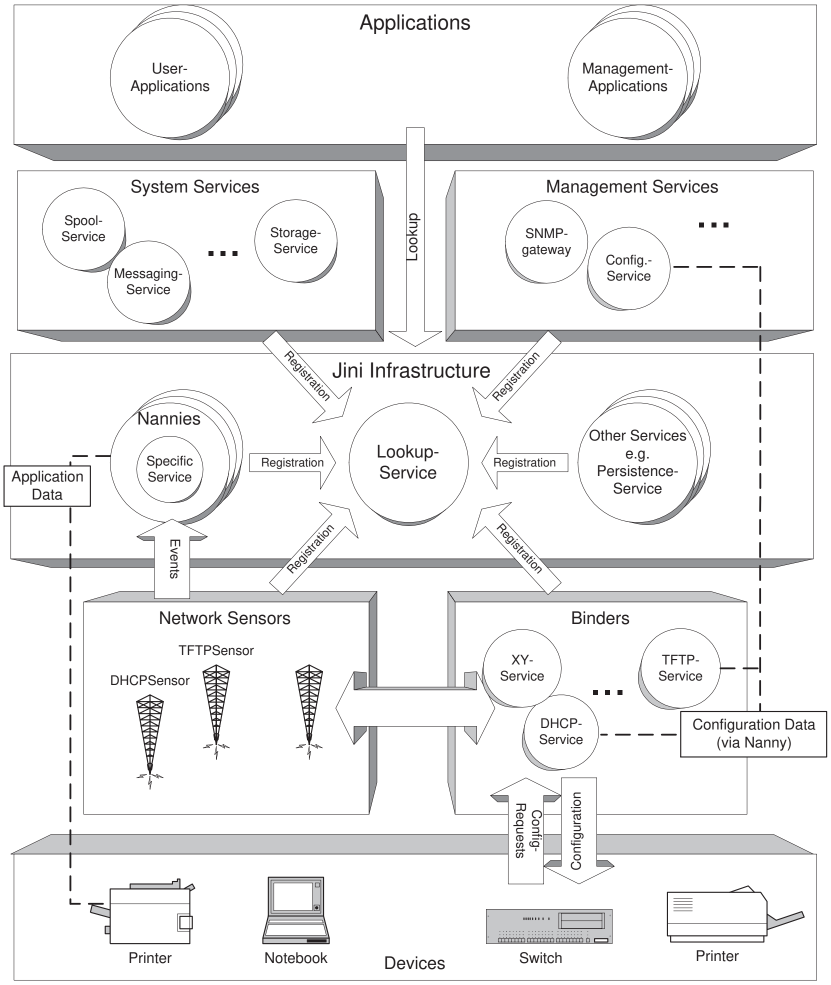

Fig. 1. Architecture for integrating legacy devices into Jini federations

Management Events. We have identified four different types of events relevant to management that can be detected by the network sensors:

– *DeviceUnknown* informs about new devices. This event is raised if a new device (e.g., one that does not yet have an IP address assigned) contacts a binder. At this time only basic information such as a MAC address might be available.
– *DeviceConnect* informs about established bindings of devices. This event might indicate that a device has successfully run the DHCP protocol and has obtained its configuration. We can assume that the device is operational with respect to the semantics of the sensor that raised the event.

– *DeviceDisconnect* informs about a device that has disconnected from the network in a proper way, e.g., by releasing an allocated IP number.
– *DeviceTimeout* indicates a failure situation caused by some timeout mechanism. It could, for example, indicate that a DHCP renewal failed.

One should keep in mind that these events are abstract in some sense and that a particular sensor needs to map concrete events to the events listed above. We think that the number of management-related events should be small and their semantics as simple as possible. Otherwise one needs to code too much sensor-specific knowledge into the interpretation of events.

Nannies. We propose a general service that registers with all available sensors to receive management-related events. Additionally, if dynamic configuration of services is needed, this component also acts as the *configuration provider* of this service. It listens for management events, connects to the configuration system in the background to query configuration information, and instantiates Java objects called *nannies* that present newly detected devices to the Jini federation and therefore to management applications.

Nannies combine device-specific knowledge, such as the native device protocol, with domain-specific knowledge, e.g., in the form of local policies defined by system administrators, in order to seamlessly integrate devices into a Jini federation. They integrate the following aspects of the device–network relationship, each of which may be optional, depending on the nature of the particular device:

– Guiding a device through its bootstrapping phase by acting as a gateway to some configuration management system.
– Receiving and propagating notifications from the sensors when devices are disconnected, e.g., caused by cancelling a DHCP lease.
– Providing the proxy object to register a service with the lookup service and renewing the leases bound to that registration.
– Implementing administrative interfaces of the Jini API, e.g., `net.jini.admin.Administrable` and `net.jini.admin.JoinAdmin`.

Nanny instantiation is crucial in some cases. In our approach nannies are instantiated with the help of a *factory* that is informed by the binders in response to devices requesting configuration information. The *nanny factory* creates a concrete nanny to handle further requests. Depending on the information available from the binder, the nanny factory instantiates a concrete device-specific nanny that knows about the kind of device it must take care of.

It should be clear that in a Jini environment the nanny does not need to implement all its facets on its own. In practice, a nanny queries the local lookup service for services that perform some tasks on its behalf. Typical services that could be "out-sourced" are gateway functionalities, lease renewals, services that provide configurable proxy objects to be stored in the lookup service, universal execution platforms for clients, etc.

A nanny could, for example, configure the device for SNMP management (if the device is SNMP enabled) by associating it with the SNMP gateway mentioned above as a manager and trap sink. It could register itself or other objects as an event listener with the gateway and react upon device specific problems according to the local policy.

In summary, a nanny takes care of a particular device in various situations – as its name suggests. From either perspective it is the layer that glues a device to its Jini federation on the one hand, and to the management federation on the other hand.

5 Example

A concrete example should demonstrate the use of our framework. We will look at what happens when a (non-Jini) printer is plugged into the network. All other components are assumed to be already running. An overview of the scenario is given in Fig. 2.

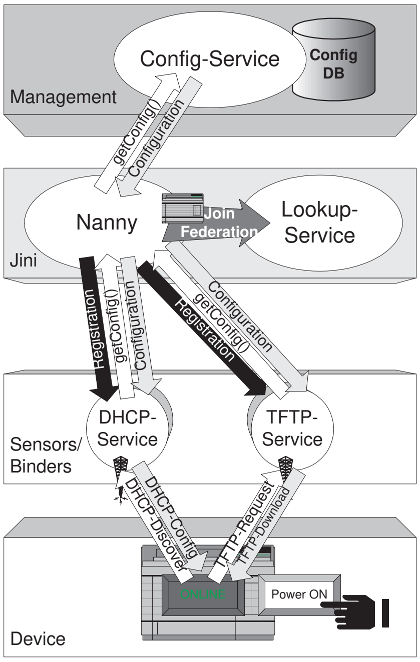

Fig. 2. Example usage scenario with a legacy printer device

The first thing a printer does after being switched on is to emit a DHCPDISCOVER message. The DHCP service receives this request and asks the nanny factory for an appropriate nanny object. Such an object may either already exist or it must be constructed by the factory. The DHCP binder forwards the request to the nanny which, in turn, may request the current configuration information from its configuration service. The configuration service returns the appropriate configuration to the nanny, which in turn delivers it to the DHCP service. This service proceeds with the DHCP protocol and offers the configuration to the requesting device.

This is rather straightforward on an abstract level but the details are trickier. The only information that the DHCP service definitely knows about the device is its media access control (MAC) address. The DHCPDISCOVER message might contain further information, but this is not mandatory. From this information, the nanny factory has to make available a device-specific nanny. As already noted, this is not always possible at the time a device is detected and must be adapted on demand as new information about the device is revealed.

In our example we assume that the nanny knows which kind of printer it is dealing with (in fact we have tested it with an HP LaserJet/JetDirect printer). It thus knows which information the printer needs and which steps it performs to retrieve this information. In our case, the printer will request its configuration profile via TFTP after accepting a DHCP configuration. Therefore, the nanny waits until it gets the *DeviceConnect* event from the DHCP sensor which means that the printer has accepted the offered configuration.

The next step is transferring the configuration profile via TFTP. The printer asks its TFTP server for a specific file. The network address of the TFTP server and the file name is included in the DHCP configuration and is supplied by either the configuration service or the nanny, depending on the local policy. Since the TFTP server is Jini-enabled (it is actually a TFTP Jini-*service*), it contacts the responsible nanny (via the factory) and asks for the printer's configuration. The nanny may already know all the configuration data or ask the configuration service to supply the missing parameters. Having acquired all information, the nanny returns it to the TFTP service which then forwards it to the printer by dynamically generating a profile by inserting the current parameters into a device specific template.

As soon as the printer received its data, the TFTP service notifies the nanny (as well as other registered event listeners) by sending a *DeviceConnect* event. The nanny knows that the printer is now fully configured and ready to accept print requests. It therefore registers a printer proxy object on behalf of the printer with the lookup service. This proxy object implements an appropriate printer interface. The printer is now a full member of the Jini federation and may be further associated with other services, e.g., a spool service, which might be subject to automatic configuration itself.

Besides providing the printer interface, the nanny should offer the Jini administration interfaces, as well as interfaces which allow management applications to perform common management tasks. Depending on the device, this task can be achieved with the help of other Jini-enabled management services. In the case of our printer it is associated with an SNMP gateway as manager and trap sink. However, the generic gateway does not know the semantics of a trap like "out of paper" and can only translate it to a generic event. Thus, the nanny registers as event consumer with the gateway, receives these events, maps them to device-specific events, and finally forwards them to interested listeners.

6 Related Work

To our knowledge there is no architecture similar to the Jini technology. Therefore we can only compare our approach with other ideas which partially cover the same domain.

The HAVi specification [5] focuses on interoperability and integration of home audio and video appliances. It defines an architecture, which is comprised of device abstraction models, software elements, protocols, an addressing scheme for devices, and a lookup service for registration of services and their resources. In this sense HAVi is comparable to Jini. In comparison to Jini though, devices are classified into four different categories ranging from devices with full HAVi support, including a Java VM, to legacy devices that use proprietary protocols only. The latter group require appropriate HAVi devices to act as gateways to the rest of the HAVi architecture. Due to the restricted application domain and the small number of devices considered, integrated management is not dealt with in the specification.

Heilbronner et al. [10] describe a DHCP server that has been equipped with sensors to send SNMP traps in response to DHCP lease state changes. They have developed an appropriate SNMP management information base (MIB) for DHCP services. Their approach integrates DHCP services into a network management environment, but they are mainly interested in monitoring aspects and it is unclear, whether further management actions could follow DHCP state changes.

Sun itself has announced a product suite to enable Java-based management (JMAPI [16]), which is a full-blown new architecture designed to integrate Java-based distributed services as well as already established management architectures. Thus it could also cope with Jini but does not directly deal with bootstrapping and initial configuration of network devices in its current specification. However, up to now it is mostly an announcement. It is planned to be a family of several products, such as the Java Dynamic Management Kit (JDMK [15]) which is already available. JDMK is primarily targeted to extend Java services by appropriate management facilities and to integrate them as managed objects into management architectures, namely SNMP.

Marzullo et al. [11] introduce the notion of sensors that provide data about the current state of applications. Contrary to our event-driven approach, control is mainly based on continuously polling the sensors. They also describe a *LanManager*, which – like a Jini lookup service – allows binding of software components at run-time. Furthermore, it can detect failed components and automatically restart them. Configuration management is not considered, though.

7 Conclusion and Future Work

We have proposed a framework for integrating legacy devices into a Jini-enabled management environment. The framework seems to be applicable to a large number of different scenarios, devices, and communication protocols. It opens up the mass of legacy devices for integration into a Jini world.

From a management perspective the advantages of our approach are visible in the concentration of all device-relevant knowledge in one dedicated nanny for each device, which seamlessly integrates the device into the local infrastructure without the administrator's intervention.

Nannies can be seen as a paradigm shift from functionality-driven services to object-centered services providing better integration of different functional management aspects. We see the future of management systems in the dynamic aggregation of specialized services in an integrated and flexible manner to overcome the existing, mostly monolithic, systems. In order to realize such management infrastructures the following still needs to be done:

- Appropriate service decompositions, including service interaction protocols, must be found.
- Standardized interfaces to management applications must be defined.
- Policies for distributed coordination, especially in failure situations, must be developed and appropriate policy enforcement subsystems must be available to interested services.
- Working systems beyond prototypes must be implemented.

Solving these problems allows us to envision a future in which nanny factories automatically download device-specific nannies from vendor Web sites and those nannies configure devices in the background according to local policies.

Acknowledgements

We would like to thank Ron Bourret, Alejandro Buchmann, and Friedemann Mattern for proof-reading and discussion of earlier drafts of this paper.

References

[1] Gerd Aschemann and Roger Kehr. Towards a Requirements-based Information Model for Configuration Management. In *Proceedings of 4th International Conference on Configurable Distributed Systems*, pages 181–189. IEEE Computer Society Press, May 1998.
[2] 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. Springer-Verlag, February 1999.
[3] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. Simple Network Management Protocol (SNMP). Internet RFC 1157, May 1990.
[4] CCITT. *Recommendation X.711, Common Management Information Protocol Specification for CCITT Applications*. CCITT, 1991.
[5] HAVi Consortium. *The HAVi Specification: Specification of the Home Audio/Video Interoperability Architecture Version 1.0 beta*, November 1998.
[6] R. Droms. Dynamic Host Configuration Protocol (DHCP). Internet RFC 2131, March 1997.
[7] Ross Finlayson, Timothy Mann, Jeffrey Mogul, and Marvin Theimer. A Reverse Address Resolution Protocol (RARP). Internet RFC 903, June 1984.
[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[9] J. Gilmore and W. J. Croft. Bootstrap Protocol (BOOTP). Internet RFC 951, September 1985.

[10] Stephen Heilbronner, Alexander Keller, and Bernhard Neumair. Integriertes Netz- und Systemmanagement mit modularen Agenten. In *Proceedings of Workstations und ihre Anwendungen SIWORK '96, Zürich, Switzerland*, 1996.

[11] K. Marzullo, R. Cooper, M. Wood, and K. Birman. Tools for Monitoring and Controlling Distributed Applications. *IEEE Computer*, 24(8):42–51, August 1991.

[12] OMG. *CORBA Object Trader Service*, December 1997.

[13] Morris Sloman, editor. *Network and Distributed Systems Management*. Addison-Wesley Publishing Company, 1994.

[14] K. R. Sollins. The TFTP Protocol (Revision 2). Internet RFC 783, June 1981.

[15] Sun Microsystems Inc. *Java Dynamic Management Kit*.

[16] Sun Microsystems Inc. *Java Management API*.

[17] Sun Microsystems Inc. *Jini Device Architecure Specification – Revision 1.0*, January 1999.

[18] Sun Microsystems Inc. *Jini Discovery and Join Specification – Revision 1.0*, January 1999.

[19] Sun Microsystems Inc. *Jini Lookup Service Specification – Revision 1.0*, January 1999.

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