A Service Gateway for Networked Sensor Systems

Shaman, an extendable Java-based service gateway for networked sensor systems, integrates small network-attached sensor-actuator modules (SAMs) into heterogeneous, high-level networking communities. The system unburdens its connected SAMs by transferring functionality from the SAMs to the gateway.

Peter Schramm, Edwin Naroska, Peter Resch, Jörg Platte, and Holger Linde
University of Dortmund

Guido Stromberg and Thomas Sturm
Infineon Technologies

An emerging area in ubiquitous computing is networked sensor systems.[1] The typical approach is to connect sensor-actuator devices using classic network infrastructures at a low level. A promising new approach is to integrate them into high-level, ad hoc networking communities. These networks can serve as infrastructures to dynamically integrate sensors and actuators into complex interactive systems while providing convenient services and interfaces to users.

A prominent scenario for ubiquitous computing and ad hoc networking is an in-house environment using smart sensor systems.[2,3] Equipping household appliances with networked sensor technology provides device-dependent services. The terminals to access these services can be devices like mobile phones or PDAs. However, invoking such services requires a negotiation between servers and clients using middleware systems.[4] Consider a small device, such as a light switch, in an in-house environment. Remote, ad hoc control of this switch requires that a client discover and a server publish its service. Both discovery and publishing must comply with the same middleware standard.

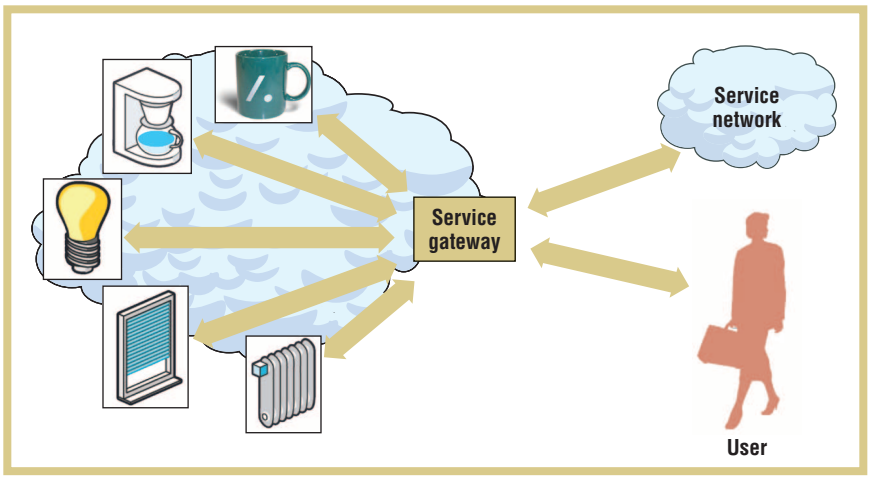
To overcome the heterogeneity of typical ubiquitous computing environments, most middleware systems are platform independent. Unfortunately, these systems make great demands on the participating devices. Consider the Jini technology (www.sun.com/jini). The key requirement of Jini's platform independence is the use of Java (http://java.sun.com).[5] Because Java requires a virtual machine (VM) with a huge memory footprint,[6] only devices with significant resources can join Jini communities. Other middleware approaches, such as UPnP (www.upnp.org), have relatively high demands on participating devices.

Using middleware systems to connect networks will soon be easier for many terminal devices. For example, several mobile phones and PDAs already have fast processors, as well as memories of 32 Mbytes, 64 Mbytes, or even 128 Mbytes. However, the situation is different for sensors. Ubiquitous computing requires integrating networked sensor technology into virtually any device, including low-cost, battery-powered ones. To achieve market success, these devices typically require extremely small, inexpensive sensor-actuator modules (SAMs) containing low-performance microcontrollers with only a few kilobytes of memory. But for services in heterogeneous environments to be convenient, they must join ad hoc networks and support middleware systems such as Jini or UPnP. Hence, these services must execute Java VMs or parse and generate XML messages.

In some cases, it might be reasonable to implement devices that directly meet all the requirements for supporting middleware systems and for

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/04/$20.00 © 2004 IEEE

Figure 1. The gateway connects the embedded world to the Internet.



joining ad hoc networking communities. However, an absolute minimization of a device's size, price, and energy consumption is often essential. Examples of such cases include networked sensor systems. Techniques are necessary that integrate extremely small network-attached SAMs into complex environments for ad hoc networking. One approach is to break down the functionality of existing protocols such as Jini to decrease their resource demands.[7,8] The significant drawback of this approach is that the modified protocols often have less functionality and provide only limited compatibility to the original protocols. The other approach is to use network proxies to execute on machines with enough resources, and then let limited devices join networking communities by acting as representatives, as the Jini architecture specification suggests (www.sun.com/software/jini/specs/devicearch1_2.pdf). However, using simple static proxies, which a system administrator must manually execute and which then run permanently, is not efficient for ad hoc networking. Instead, a proxy should automatically begin and shut down on demand. Shaman, a Java-based service gateway, meets these requirements.

The Shaman concept

Shaman (taken from a word that typically refers to a mediator between two types of beings, humans and spirits) is an extendable, scalable service gateway that uses network proxies to integrate extremely small networked SAMs into heterogeneous ad hoc networking communities. Thus, SAMs can provide and use high-level Web services that comply with common middleware standards such as Jini. Figure 1 illustrates this for the in-house scenario.

The system also contains a mapping of GUI modules so that users can use a SAM's service directly, either for administrative purposes or for direct-service use. The following explains other key features in our system.

Multiple service interfaces

The gateway system's infrastructure isn't tied to one particular standard for ad hoc networking. It can easily work with nearly any standard using *service wrappers*, special modules containing standard-specific functionality. Thus, if the gateway has the appropriate service wrappers, a SAM can act as a Jini and an UPnP server simultaneously. Hence, clients can choose a compatible service interface.

The current system supports the following types of service interfaces:

- A Jini interface for the integration of SAMs into Jini communities. Thus, SAMs can provide Jini services and be clients of other Jini services, forming interconnected sensor services.
- An interface using Java applets to provide SAMs with GUIs. This serves an administrative purpose and, where applicable, is for direct-service use by a human.
- An HTML interface so that clients that are not Java-enabled, or are too restricted to execute the GUI module, can access a SAM's service.

Users can also configure the system to send UPnP basic-device announcements for each connected SAM. Hence, UPnP-compliant systems can easily discover SAMs. (The UPnP presentation pages link to the system's Java applet interface. Using PDAs with systems such as Pocket PC 2003, users only need to double click on a discovered SAM to invoke the GUI.)

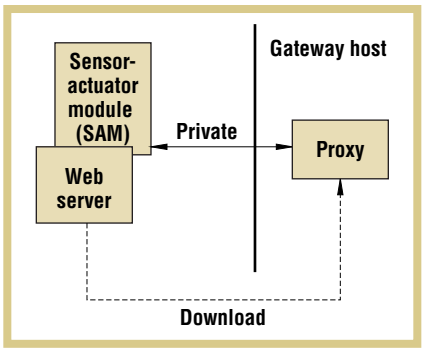


Figure 2. The proxy acts on behalf of the sensor-actuator module (SAM) and uses the gateway host's resources.

Smart-driver concept

To simplify deployment of new services, we introduce an easy-to-use driver concept. The driver is a downloadable software component containing two modules: the network proxy that acts on behalf of the SAM and an optional GUI module for manual control and administration of a SAM's service. The driver may be from any Web server that the gateway can reach. However, as Figure 2 shows, the SAM itself may embed a Web server that provides the driver. Thus, unlike typical approaches, this approach lets the SAM itself inject dri-

ver software into the system. Hence, there is no need for manual software installation.

The proxy's main objective is to translate client requests into SAM-specific commands. The proxy can use a private, highly optimized protocol to communicate with its SAM. Thus, the proxy is a SAM's front end for providing and using services in the network. Because SAMs and proxies form distributed systems, SAMs can easily outsource functionality to their proxies to use the gateway host's resources. Therefore, a sensor needs to provide only the raw data; developers can implement all kinds of data processing and everything that makes the sensor smart in the proxy using the object-oriented Java programming language's full functional range.

Furthermore, the proxy plays an essential role for SAMs with restricted energy resources, such as battery-driven devices. Because the proxy is the actual service provider, the SAM can enter standby mode while the proxy stays online. In many cases, it's sufficient for the SAM to wake up periodically to update the proxy with new state information. Therefore, the proxy can immediately answer any service request even if the SAM is in power-down mode.

One GUI, *x* interfaces

For cases requiring a GUI, a service developer only need provide a single GUI module based on Java Swing that automatically maps to each kind of client interface (Java applet, HTML, and Jini, as appropriate). The advantage of our one-GUI, *x*-interfaces concept is that it minimizes design effort and automatically gives the same look and feel to users of different interfaces. Hence, users can choose an appropriate interface type based on the terminal device's capabilities. For example, even though most PDAs lack Java VMs, the HTML interface lets them still serve as terminals.

Plug-and-work capabilities

An important aspect for the gateway is the ability to provide services without any manual configuration efforts. We achieve this using service attributes and leasing techniques. When connecting to the gateway, a SAM must describe its service by submitting service attributes within a special boot protocol. These attributes contain all the information the gateway needs to provide a service on behalf of the SAM. Examples of service attributes are *service name*, *service vendor*, *SAM version*, and location information. The boot protocol contains a simple discovery mechanism that lets a SAM find a gateway to deploy its proxy.

Integrating SAMs into the gateway system requires a certain amount of resources from the gateway host. Allocating these resources only on demand is reasonable. The SAM must lease these resources from the gateway by submitting a desired leasing period within the boot protocol. The gateway will represent a service on behalf of the SAM only for this period of time. If the SAM doesn't send a *proof of life* to the gateway before the leasing period expires, the gateway will shut down the corresponding service, assuming the SAM is no longer online.

The proxy's main objective is to translate client requests into SAM-specific commands. The proxy can use a private, highly optimized protocol to communicate with its SAM.

Scalability

For fault tolerance, multiple gateways are necessary. Using gateways is easy because they provide a kind of native load balancing. A SAM connects to the first gateway that responds to its discovery requests. In most cases, this is the gateway with the lowest load and the shortest network latency. However, using multiple gateways doesn't necessarily add costs; existing Java-enabled appliances can serve as gateway hosts. Consider a Java-enabled TV box or the frequently cited Java refrigerator that can act as gateways for in-house sensors.

The Shaman alarm system scenario

We now present a scenario that demonstrates how to apply Shaman technology to a hotel that needs a smart and reliable fire alarm system. We chose this scenario because it exemplifies the system's features and addresses some important aspects of ubiquitous computing and sensor networks, including

- *Energy efficiency.* Most sensors in ubiquitous computing environments have restricted energy resources. Some are even battery powered.
- *Fault tolerance.* Failures of single components are inevitable in complex systems. However, their detection and correction is essential.
- *Sensor fusion.* Letting sensors query one another to increase flexibility and avoid dispensable hardware costs is reasonable.
- *Configurability.* Providing convenient tools to configure complex environments and store configurations is essential.
- *Human-machine interfaces.* Humans play the central role in almost all ubiq-

uitous computing scenarios, especially those involving human safety.

For all our examples, we use Jini technology as our reference for smart services and ad hoc networking. The Shaman gateway system is the enabler that brings the embedded world and Jini together. We begin with the deployment of the Shaman alarm system in the hotel building. To provide fault tolerance, we abandon any notion of centralized control, preferring instead a distributed alarm system. For the same reason, we deploy multiple gateways—enough to provide all wireless sensors with a gateway within range. Furthermore, we deploy wireless Shaman-enabled temperature sensors, as well as smoke and gas detectors in each hotel room and in the basement. Each floor has alarm bells and fire extinguishers. In the hotel rooms, we do without dedicated alarm signaling devices, because existing appliances such as TVs, hi-fi systems, and room lighting can serve as alarm transmitters that either are Shaman enabled or come with native Jini support.

Handling configuration problems

Compliant with Shaman technology, all deployed SAMs send discovery messages and deliver their proxies to the discovered gateways. The gateways register Jini services for each SAM. But there are cases when an administrator must configure services after a SAM has been attached to the network. (For example, a smoke detector's developer can't know the name of the room in which it's deployed, so an administrator must manually set this location.) Administrators can configure these services using their PDAs to access each SAM's applet-based administration interface. The administrators set information such as service name and service location—for example, "smoke detector, room 112." Because most SAMs have no flash memory, each proxy uses a Jini-based storage service to save its configuration under a unique hardware ID. On the next startup—for example, after restoring a drained battery—the proxy automatically fetches the correct data without manual configuration. After configuration, the system is on duty and performing fine.

Temperature monitoring

In the mean time, let's check the temperature sensor in the cellar. Internally, it obtains a voltage value from a thermocouple, and its only task is to periodically transmit this voltage value to its proxy. The proxy must then calculate a temperature from the voltage value. Because most parts of a sensor's service are implemented in the proxy, it is easily extendable. Hence, the proxy can switch from Fahrenheit to Celsius, for example. In standard mode, the temperature sensor has a low duty cycle; it measures new temperature values every 5 minutes. Because the sensor is battery driven, it must save energy. Hence, it enters standby modes between measurements. In the classic approach, no one could poll a stand-alone sensor while it's offline. In the proxy approach, however, the temperature service is always available, even when the sensor is in standby mode.

Self-diagnostic capabilities

The system also has self-diagnostic capabilities. Imagine that all sensors use the Jini-based storage service just mentioned to log their measured values. Let's say that a diagnostic routine in the proxy of one of the temperature sensors detects that the measured values haven't changed for a while. To see if something went wrong, the proxy consults another temperature sensor nearby using the Jini lookup mechanisms. Comparing the two sensors' temperature curves and noticing significant differences, the proxy concludes that a malfunction might have occurred. So, it automatically sends a remote event to the administrator to replace the faulty sensor.

In another case, failure diagnostics is easier. This time, one of the smoke detectors fails completely. As soon as its leasing period expires, its proxy automatically shuts down, and its service is deregistered. In addition, the system fires an event to call the administrator for replacement. If a gateway fails completely, the connected SAMs recognize the failure as soon as they try to renew their leases. Because the gateway doesn't confirm the lease renewal, the SAMs try to find other gateways to connect to.

Full-scale alarm procedures

Suddenly a smoke detector on the 13th floor detects smoke and sounds off. The SAM immediately informs its proxy, which then uses Jini technology to contact the proxy of a temperature sensor in the same room. Detecting a significant temperature rise, the proxy decides to raise an alarm at the "danger" severity level because it assumes a fire has broken out. The proxy uses Jini technology again to fetch all services that implement a special alarm interface. These devices can react to an alarm by various means. Using this interface, the proxy informs all devices about the alarm, the severity level, what has happened, and where. Of course, the Shaman-enabled fire extinguishers in each room and alarm bells on each floor implement the alarm interface. But the Jini- or Shaman-enabled TVs, hi-fi systems, and other appliances also implement the interface. Hence, all appliances in the hotel aid the evacuation of the building. The lights in every occupied room automatically turn on. While the alarm bells sound, all devices with displays such as TVs show the message "Danger! Fire Alarm!" and show escape routes, and the hi-fi systems play alarm sounds.

Alternative Network Sensor Approaches

Here we discuss some other approaches for handling networked sensor systems. In addition to our approach, other projects deal with the integration of limited devices into ubiquitous computing environments. Some projects prefer gateways and use network proxies, as we do. Others focus on direct network connections and the associated problems, such as routing, resource allocation, and energy consumption.

Jini Surrogate Architecture

A framework that provides dynamic, plug-and-work lifecycle management for Jini-based network proxies is Sun's Jini Surrogate Architecture (http://surrogate.jini.org). Although it integrates non-Jini devices into Jini communities, this architecture has substantial disadvantages regarding typical environments for ubiquitous computing. First, programmers who want to let small servers join a Jini network must write complete network surrogates. Thus, to provide convenient services, they must implement full-featured Jini servers with adequate client, user, and session support that fit into the surrogate framework. Second, this architecture is restricted to Jini networks. This is a drawback for heterogeneous environments containing clients that comply with different standards of ad hoc networks.

OSGi-compliant gateway solutions

The Open Services Gateway Initiative (OSGi) has specified a framework for general-purpose service deployment.[1] OSGi-compliant gateways serve as bridges between local/home area networks and the Internet. The OSGi Service Platform is an execution environment for remotely deployed services. Users can add new service applications using installation packages known as bundles. However, an OSGi-compliant system is not self-contained, because an external instance called a gateway operator must install these bundles.

ProSyst describes an OSGi compliant end-to-end solution.[2] The system includes basic services for smart-home environments and packages to integrate other protocol standards such as UPnP or Jini. Thus, users can access UPnP-compliant devices through the OSGi framework. However, the OSGi framework doesn't aim to unburden small devices in local network environments and thus covers another domain, as we do with our approach. Nevertheless, it is possible to combine both technologies—that is, to deploy our gateway system as a bundle in an OSGi framework.

Routing optimization approaches

There has been considerable work in the field of sensor networks, where one major focus is on optimizing routing algorithms to decrease network congestion and power consumption.[3,4] Our system assumes that a lower network layer instance has already solved the routing problem.

Resource allocation in sensor networks

Because processing sensor data often requires significant computational resources, researchers have also addressed the allocation of these resources in the network. Some architectures exploit the sensor modules' processing power,[4–7] while others use dedicated high-performance hardware nodes to collect and process data received from sensors.[8,9] The second type of architecture resembles our approach in that both concern dedicated nodes

Architecture and implementation

Here we describe the architecture and some implementation details of our gateway system. (See the "Alternative Network Sensor Approaches" sidebar for a comparison of our system with some other frameworks.) Figure 3 shows the general structure. Figure 4 shows a single gateway service that negotiates communication between a SAM and three types of clients. We implemented the system using Java because it supports many standards (such as Jini) for distributed computing and ad hoc networking. Besides platform independence, Java programs can load and instantiate code during runtime. This is very important for dynamic downloading and instantiation of driver classes.

When we say "the gateway," we mean the host platform on which one or more gateway services execute. Each gateway service is a process (running in its own Java VM) that belongs to one SAM and can handle multiple clients. (Although it increases demands on resources, deploying gateway services in separate VMs is advantageous because defective gateway services can't compromise the entire system. Furthermore, resources can easily be allocated and deallocated on demand.) Hence, each SAM is associated with its own gateway service.

The gateway system's core is the *service manager*, which lists all currently connected SAMs in an internal registry. To advertise its service, a SAM must connect to the service manager using the boot protocol. Following the leasing concept, the service manager boots gateway services on demand and shuts them down when they are no longer needed.

Gateway service

The gateway service's inner components are as follows. To overcome the problem of a SAM not being able to handle multiple client connections, the gateway service stores client requests in a *request queue*. The system sequentially passes these requests to a SAM proxy. The SAM proxy is part of the gateway service, whereas the GUI module is

that assist the sensors. Lim's work, in particular, addresses similar problems.[9] However, unlike our technique, these architectures don't solve the problem of dynamically integrating (resource) limited sensors into the network during runtime and making them available to high-level ad hoc services. We expect future sensor networks to be heterogeneous and highly dynamic, frequently installing new modules and removing defective hardware. Consequently, the network must seamlessly integrate sensor types that the system doesn't know in advance. Moreover, to enhance usability, these networks must connect to modern ad hoc network infrastructures. Finally, such approaches don't efficiently provide user interfaces on demand for users or administrators. In contrast, Shaman provides a convenient, easily implemented solution to all these problems.

Network proxies with user interface adaptation

Many other projects also use network proxies but focus primarily on adapting information presentation and user interfaces to client needs and abilities.[10–12] The major difference in our approach is that we focus on both the client and the server.

REFERENCES

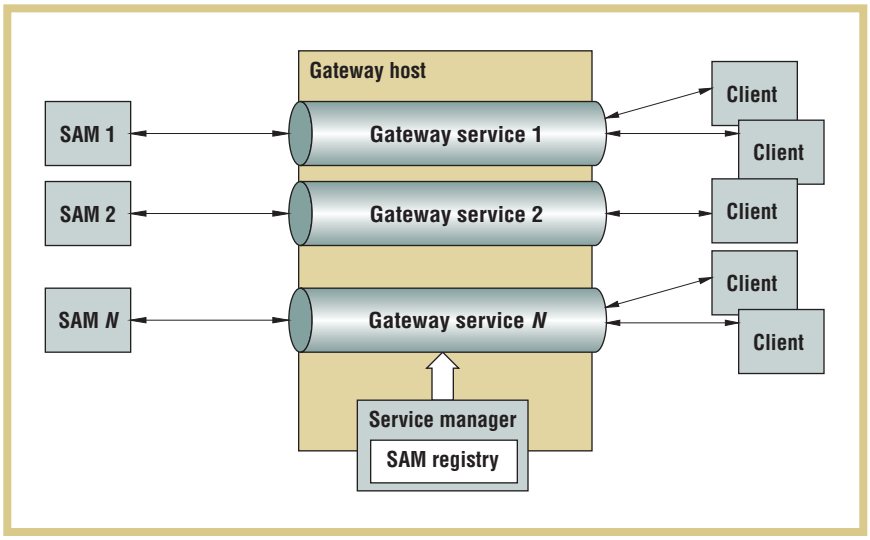
1. D. Marples and P. Kriens, "The Open Services Gateway Initiative: An Introductory Overview," *IEEE Comm.*, vol. 39, no. 12, Dec. 2001, pp. 110–114.
2. D. Valetchev and I. Frankow, "Service Gateway Architecture for a Smart Home," *IEEE Comm. Magazine*, vol. 40, no. 5, Apr. 2002, pp. 126–132.
3. J.A. Stankovic et al., "Real-Time Communication and Coordination in Embedded Sensor Networks," *Proc. IEEE*, vol. 91, no. 7, July 2003, pp. 1002–1022.
4. Y. He et al., "A Programmable Routing Framework for Autonomic Sensor Networks," *Proc. 5th Ann. Int'l Workshop Active Middleware Services: Autonomic Computing Workshop* (AMS 03), IEEE CS Press, 2003, pp. 60–68.
5. T. Liu and M. Martonosi, "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems," *Proc. 9th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPOPP 03), ACM Press, 2003, pp. 107–118.
6. P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 85-95.
7. R. Barr et al. "On the Need for System-Level Support for Ad Hoc and Sensor Networks," *Operating Systems Rev.*, vol. 36, no. 2, pp. 1–5.
8. L. Subramanian and R. Katz, "An Architecture for Building Self-Configurable Systems," *Proc. IEEE/ACM Workshop Mobile and Ad Hoc Networking and Computing* (MobiHOC 00), IEEE Press, 2000, pp. 63–73.
9. A. Lim, "Support for Reliability in Self-Organizing Sensor Networks," *Proc. 5th Int'l Conf. Information Fusion*, IEEE Press, 2002, vol. 2, pp. 973–980.
10. P. Maniatis et al., "The Mobile People Architecture," *Mobile Computing and Comm. Rev.*, vol. 3, no. 3, July 1999, pp. 36–42.
11. S. Ross et al., "A Composable Framework for Secure Multi-Modal Access to Internet Services from Post-PC Devices," *Mobile Networks and Applications*, vol. 7, no. 5, Oct. 2002, pp. 389–406.
12. T.D. Hodes and R.H. Katz. "Composeable Ad Hoc Location-Based Services for Heterogeneous Mobile Clients," *ACM Wireless Networks J.*, vol. 5, no. 5, Oct. 1999, pp. 411–427.

exported by the gateway's Web server for client download. The service wrapper communicates with one or more clients and preprocesses their requests. Then it hands the requests over to the request queue. Besides the option to implement multiple service interfaces in one wrapper, it's possible to install more than one service wrapper at the same time to simultaneously support multiple standards of distributed computing.

Client interfaces

So far, our service wrapper provides the Jini, applet, and HTML interfaces, and the basic UPnP announcer. The key element of the Jini interface (see client type A in Figure 4) is a Jini service object that the gateway automatically provides.

Figure 3. The gateway host running several gateway services, and the service manager.



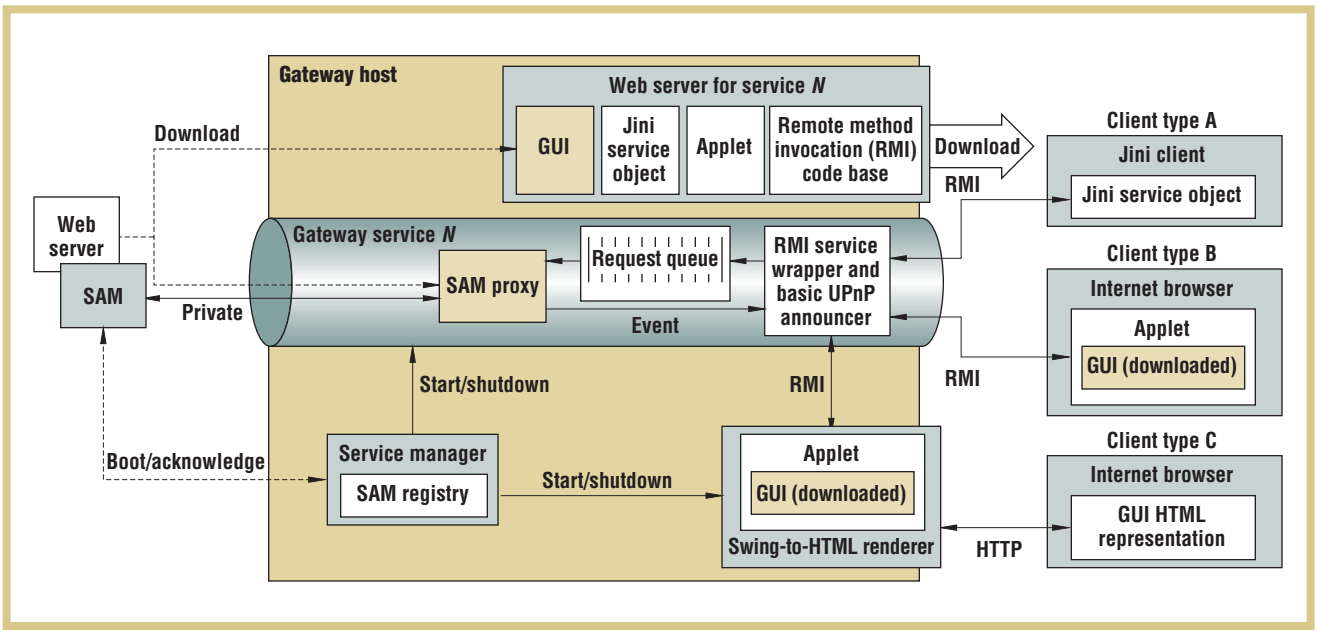


Figure 4. A gateway service connecting a SAM to three different types of clients.

The applet interface (client type B in Figure 4) is intended solely for use with a corresponding GUI object. Similar to the Jini service object, the applet code is automatically provided by the gateway. Once the applet is running, it downloads the GUI object from the gateway and instantiates it. Then it opens a remote-method-invocation (RMI) connection to send requests to the service wrapper.

In addition to the applet interface, the wrapper provides an HTML interface to control services (client type C in Figure 4). The corresponding HTML code doesn't require manual implementation. Instead, the system extracts this code from the Java applet's code dynamically. We achieved this using CreamTec's Swing-to-HTML converter, WebCream (www.creamtec.com/webcream). WebCream can execute a Java applet in a container application on a server similar to a Java servlet. A special renderer class maps all Swing functionality to HTML. Hence, the HTML interface uses exactly the same application code as the applet interface but uses HTML instead of Swing for visualization. Because the applet executes on the gateway rather than on the client, and the client has to handle only HTML code, this approach unburdens resource-restricted clients.

Each gateway service contains a component called a basic UPnP announcer, which publishes a SAM service as a UPnP basic device. Such a device has no methods but provides the URL for a presentation Web page. Because we use the presentation page to give users a service's Java applet, they can easily invoke services without knowing the discovery Web page's address.

Implementing a new service

Deploying a completely new service requires implementing the following components:

- *SAM*. The SAM must communicate using a gateway-compatible boot protocol. Moreover, it must implement the server part of the private communication protocol between itself and the SAM proxy.
- *SAM proxy*. This standard Java class implements the client part of the private communication protocol. It must also implement the appropriate interfaces to fit into the gateway framework. Figure 5 gives an example of a SAM proxy.
- *GUI module* (optional). The GUI module is a Java Swing class that implements the user GUI. Note that the GUI presentation automatically maps to the different service interfaces used by standalone Jini applications, Java applets, and HTML-based clients. Thus, the service developer needs to provide only a single GUI implementation.

Experiments

Our experimental setup included the gateway host, some SAMs, and various clients of all supported types. For the gateway host, we used a standard PC with a HotSpot VM from the Java 2 Standard Edition. We executed the SAMs in native code on DIL/NetPCs. Communication with the gateway used a UDP/IP (user datagram protocol, Internet protocol) Ethernet connection. As terminal devices, we used Bluetooth-enabled Compaq iPaqs and the NSICom CrEme Personal Java VM. Our example applications included the remote light switch and the control software for the alarm system (see Figure 6) mentioned earlier.

Because we decided to encapsulate each gateway service in a separate VM, memory demands were high (about 3 Mbytes

Figure 5. A code example for a SAM proxy that connects to a light switch. (This code serves only to give an idea of an implementation effort and code size; it doesn't contain any exception handling.)

```
public class SAMProxy implements SAMProxyInt, Runnable {
    int lightIntensity = 0, ServerPort = –1;
    InetAddress SAMAddress = null;

//The following method initializes SAM proxy, taking the port number from the service manager
and starting a thread to receive data from the SAM.

    public void initialize(int ServerPort) {
        this.ServerPort = ServerPort;
        Thread thread = new Thread(this);
        thread.start();
    }

//This thread waits for incoming user datagram protocol (UDP) packets from the SAM and, for
each packet received, updates the light intensity's local value.

    public void run() {
        byte[] inbuf = new byte[1024], data = new byte[1024];
        DatagramSocket socket = new DatagramSocket(ServerPort);
        DatagramPacket packet = new DatagramPacket(inbuf, inbuf.length);
        while (true) {
            socket.receive(packet);
            lightIntensity =
                Integer.parseInt(new String(inbuf, 0, 1, packet.getLength()));
            SAMAddress = packet.getAddress();
        }
    }

//This method hands a command to the SAM. The methodName string identifies the command,
whose parameters are handed over within a vector object. This method then interprets
methodName and sends a corresponding command to the SAM via UDP. At last, this method
packs the actual light intensity into a vector object and hands it back as a result parameter.

    public Vector invoke(String methodName, Vector parameterList, int clientID) {
        byte[] inbuf = new byte[1024], data = new byte[1024];
        if (methodName.equals(new String("setLightIntensity"))) {
            Object o = parameterList.elementAt(0);
            lightIntensity = ((Integer)o).intValue();
            String s = lightIntensity;
            byte[] outbuf = s.getBytes();
            DatagramPacket packet =
                new DatagramPacket(outbuf, outbuf.length, SAMAddress, 7788);
            DatagramSocket socket = new DatagramSocket();
            socket.send(packet);
        }

        Vector resultList = new Vector();
        resultList.addElement(new Integer(lightIntensity));
        return resultList;
    }
}
```

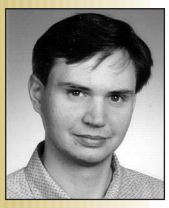
per gateway service). The service manager consumed about 8 Mbytes. Although these memory requirements are no problem for a standard PC, they are targets for future improvements. On the other hand, we achieved our SAM implementation in native C code using less than 10 Kbytes of memory footprint (not including the size of the UDP/IP protocol stack). Therefore, this implementation is suitable for microcontroller-based embedded servers, which common SAMs use. The code sizes of the SAM proxies were 3.5 Kbytes and 10 Kbytes. The first proxy used a private protocol with simple byte sequences as messages; the second implemented a serial protocol for ASCII data. The code for the GUIs consumed about 3.5 Kbytes.

To give an idea of a SAM proxy's typical code size, we can compare it to the size of a stand-alone Jini application performing tasks similar to those the proxy

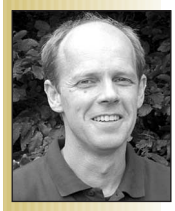
Figure 6. The Java applet to control the exemplary alarm system.

the AUTHORS

Peter Schramm is a doctoral candidate at the Computer Engineering Institute of the University of Dortmund, Germany. His research interests include ubiquitous computing and middleware systems. He received a diploma in electrical engineering from the University of Dortmund. Contact him at the Computer Eng. Inst., Univ. of Dortmund, Otto-Hahn-Str. 4, D-44221 Dortmund, Germany; peter.schramm@uni-dortmund.de.



Edwin Naroska is an assistant professor at the Computer Engineering Institute of the University of Dortmund. His research interests include design and verification of VLSI circuits and design of hardware and software for embedded systems. He received a PhD in electrical engineering from the University of Dortmund. Contact him at edwin.naroska@uni-dortmund.de.

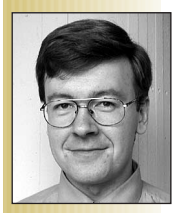


Peter Resch is an electrical engineer at the Computer Engineering Institute of the University of Dortmund. His research interests include microprocessor design and the design of mobile and ubiquitous devices and their applications. He received a graduate engineering degree from the Dortmund University of Applied Sciences. Contact him at peter.resch@uni-dortmund.de.

Jörg Platte is a doctoral candidate at the Computer Engineering Institute of the University of Dortmund. His research interests include embedded systems, and security and privacy in ubiquitous computing environments. He received a PhD in electrical engineering from the University of Dortmund. Contact him at joerg.platte@uni-dortmund.de.

Holger Linde is a doctoral candidate at the Graduate School of Production Engineering and Logistics, Dortmund. His research interests include embedded and ubiquitous computing, particularly localization methodologies. He received a diploma in electrical engineering from the University of Dortmund. Contact him at holger.linde@uni-dortmund.de.

Guido Stromberg is a research staff expert at Infineon Technologies, Corporate Research. His research interests include signal processing, coding theory, embedded systems, and mobile networking. He received a PhD in electrical engineering from the University of Dortmund. Contact him at guido.stromberg@infineon.com.



Thomas Sturm is a professor of mathematics at the University of the Federal Armed Forces, Munich. His research interests include numerical algorithms and system architectures for ambient intelligence, stochastic algorithms, decoding methods for communications, and nonlinear optimization theory. He received a doctorate in mathematics from the University of Technology in Munich, and a doctorate in electrical engineering from the University of the Federal Armed Forces, Munich. Contact him at thomas.sturm@unibw-muenchen.de.

performs with a gateway service. This includes all Jini internals (such as lookup and join) and other features (such as request queuing and event handling). The proxy code consumed 3.5 Kbytes, and the stand-alone application consumed 18 Kbytes. Thus, the savings in code size was about 80 percent.

Our results show that with service gateways, small servers with very little processing power and memory can participate in complex networking environments. Our experiments also prove that using service gateways can be convenient and efficient for heterogeneous computing environments. Now that our first prototype implementation has proven technically feasible, we hope to improve and extend the system. First, we plan to add additional service wrappers to support more protocol standards for ad hoc networking—for example, full implementation of UPnP. Second, we will implement SAMs based on proprietary hardware platforms. Finally, we plan to optimize the implementation of the gateway itself in terms of resource consumption and throughput. P

REFERENCES

1. M. Weiser, "The Computer for the Twenty-First Century," *Scientific Amer.*, Sept. 1991, pp. 94–104.
2. B. Brummit et al., "Easy Living: Technologies for Intelligent Environments," *Proc. Symp. Handheld and Ubiquitous Computing* (HUC 00), Springer, 2000, pp. 25–27.
3. W.S. Conner, L. Krishnamurty, and R. Want, "Making Every Day Life Easier Using Dense Sensor Networks," *Proc. Ubicomp 2001*, Springer, 2001, pp. 49–55.
4. S. Helal, "Standards for Service Discovery and Delivery," *IEEE Pervasive Computing*, vol. 1, no. 3, July–Sept. 2002, pp. 95–100.
5. M. Barr, "Java Technology Overview," *Proc. Embedded Systems Conf. San Francisco*, CMP Media LLC, 2002, parts 1 and 2, nos. 308 and 348.
6. M. Laukkanen, "Java on Handheld Devices: Comparing J2me Cdc to Java 1.1 and Java 2," CiteSeer, NEC Research Inst., 2001; http://citeseer.nj.nec.com/473890.html.
7. P. Huang et al., "Jini for Ubiquitous Devices," CiteSeer, NEC Research Inst., 2002; http://citeseer.nj.nec.com/537253.html.
8. V. Lenders, P. Huang, and M. Muheim, "Hybrid Jini for Limited Devices," *Proc. IEEE Int'l Conf. Wireless LANs and Home Networks* (ICWLHN 01), World Scientific Publishing, 2001, pp. 27-34.