RESTful Device Interaction Architecture for Embedded Devices

Hayakawa, Hiroshi
Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University : Graduate Student

Sato, Kenya
Department of Information Systems Design, Faculty of Science and Engineering, Doshisha University

Tagashira, Shigeaki
Department of Advanced Information Technology, Faculty of Information Science and Electrical Engineering, Kyushu University

Nakanishi, Tsuneo
Department of Advanced Information Technology, Faculty of Information Science and Electrical Engineering, Kyushu University

他

https://doi.org/10.15017/1517960

出版情報：九州大学大学院システム情報科学紀要. 14 (2), pp.53-58, 2009-09-25. 九州大学大学院システム情報科学研究院

九州大学大学院
システム情報科学紀要
第14巻 第2号 平成21年9月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.14 No.2, September 2009

RESTful Device Interaction Architecture for Embedded Devices

Hiroshi HAYAKAWA*, Kenya SATO**, Shigeaki TAGASHIRA***,
Tsuneo NAKANISHI*** and Akira FUKUDA***

(Received June 11, 2009)

Abstract: Toward ubiquitous network society, there have been challenges to construct smart spaces in living space such as homes and offices by connecting embedded devices with each other; however, many existing smart spaces tend to depend on particular platform or be a self-contained environment. In this paper, we describe requirements on frameworks for smart spaces with embedded devices, and then propose a new framework based on the requirements. The proposed framework is designed to have high scalability and be suitable for cross-platform environments by employing RESTful web service for device interaction architecture. By encoding communication messages for requesting services in x-www-form-urlencoded scheme, the proposed framework can support many kinds of devices including resource-limited devices which can't afford an XML parser. Evaluation with existing frameworks shows the proposed framework is suitable as a framework for smart spaces because of its simple and lightweight architecture.

Keywords: Embedded device, Device interaction, REST, Framework, UPnP, Jini

1. Introduction

Capability and complexity in embedded systems are getting more diverse. The capability of some high-end devices such as smart phones and PDAs is comparable to that of a PC. Whereas some of white goods and sensor devices have strict limitation in cost of development and production, thus they have a minimum capability. Toward ubiquitous network society, there have been some challenges to construct smart spaces in living space such as homes and offices by connecting these devices with each other. Some frameworks such as DLNA (Digital Living Network Alliance)[1], Jini[2),3)], HAVi (Home Audio/Video Interoperability)[4], and ECHONET (Energy Conservation and Homecare Network)[5] have been proposed for constructing smart spaces. Existing smart spaces such as Jini and HAVi tend to depend on particular platform and be a self-contained environment. UPnP employs popular Internet technologies and is suitable for cross-platform environments; however, as communication messages are described in XML, every device must have an XML parser. Considering that the available resources of embedded devices are limited and heterogeneous, the resources required for device interaction should be as small as possible in order to support many devices. Additionally, traditional RPC architecture in UPnP and Jini is not appropriate in scalability.

In this paper, we propose a new framework named SONICA (Service Oriented Network Interoperability for Component Adaptation). SONICA adopts RESTful web service for device interaction. REST (Representational State Transfer)[6] is an architecture underlying traditional HTTP scheme. As a RESTful web service employs HTTP methods as its service interface, the service is suitable for cross-platform environments. Moreover, a RESTful web service is simple and has high scalability. As communication messages of SONICA for requesting services are encoded in x-www-form-urlencoded scheme, SONICA can support resource-limited devices which can't afford an XML parser. SONICA can also collaborate with existing frameworks which have no open service interfaces by adding a proxy component.

The outline of this paper is as follows: The next section introduces related works and discusses the requirements on architecture for device interaction in a smart space with embedded devices. In the 3rd section, we show the design of SONICA. We evaluate SONICA against existing frameworks in the 4th section. In the 5th section, we show a practical system that supports IEEE 1394 devices by a proxy component, and then conclude with a summary in the 6th section.

* Department of Computer Science and Communication Engineering, Graduate Student
** Department of Information Systems Design, Faculty of Science and Engineering, Doshisha University
*** Department of Advanced Information Technology

2. Related Works

DLNA (Digital Living Network Alliance) aligns industry leaders including Microsoft, Intel, Panasonic, and SONY, and is becoming the de facto standard framework for constructing smart home spaces. DLNA adopts UPnP (Universal Plug & Play) Device Architecture 1.0[7] for device interoperability. UPnP was proposed by Microsoft who aimed to apply the concept of Plug & Play to network systems. The technologies underlying UPnP Device Architecture include SOAP[8)−11)] as a remote procedure call (RPC) mechanism. SOAP allows devices to offer complex and advanced services. However, each UPnP device must have an XML parser and must parse all communication messages described in XML. This results in inefficiency and requires additional software resources.

Newmarch[12] proposed a RESTful approach that substitutes RESTful web services for SOAP RPCs. The approach is about twice as fast as SOAP-based UPnP in making 1000 queries, and reduces query size by about 80 to 90%. He also made a comparison of data encoding manners, and showed that the size of x-www-form-urlencoded data and its parser is less than 5% of the size of XML-encoded data and an XML parser.

Jini was proposed by Sun Microsystems in 1999 aiming at a network plug & play especially for dynamic networks where resources are volatile. For the last few years, Jini has been attracting attention once again because of its new license system and improvements in the performance of embedded systems. In Jini systems, a server registers an instance of its service interface, codebase annotation and its attributes to a lookup service. Using a template consists of a service's interface and attributes, a client retrieves services through the lookup service. The actual implementation of a ServiceItem, such as an instance of service or may a proxy, will be downloaded from the location specified by the codebase annotation in the ServiceItem that the server supplied to the lookup service. In order to find services and re-serialize a marshalled ServiceItem, clients have to know the service's interface in advance. This means that clients and management systems such as a lookup service and a codebase need to have every interface of continuously upcoming services and devices, and incurs low scalability. In addition, increase in network traffic by transferring objects via network is also a concern.

Vayssiere[13] improved this issue by introducing an independent Jini service called adapter service. An adapter service employs adapter pattern in GoF design patterns so that the service brokers services which provide similar functions, although they inherit incompatible interfaces with each other. However, each service requires one by one adapter for others.

In Adaptive Jini[14] systems, a server provides the UI of its services to clients; although in conventional Jini systems, clients need to implement UIs based on each service interface. Adaptive Jini system enables users to operate a server device by downloading and running its UI on a client. However, the mechanism of Adaptive Jini is strongly depends on Java platform, and is difficult to apply other platforms.

Jini specifications are not constrained to Java; though almost all of Jini services are written in Java, and their communications are based on Java RMI (Remote Method Invocation) mechanism. Some high-end embedded devices adopt Java Platform Micro Edition (J2ME), however not all profiles support object serialization inevitable for RMI. JiniME[15] and μJini proxy architecture[16] are proposed to improve connectivity among Jini services and embedded devices under severe memory restrictions. In these technologies, devices called Jini Bridge and μJini proxy act as a proxy device for an embedded device. And, the proxy interacts with a Jini service instead of an embedded device. However, to use a new service via these proxies devices, Jini Bridge and μJini proxy have to implement the interface of the new service.

In accordance with these works, we outline the requirements for frameworks for smart spaces with embedded devices.

- Service interface for cross-platform environments: Many smart spaces have tended to be a self-contained environment. However, it is expected that collaborations among heterogeneous smart spaces are getting more common. To realize seamless connection in such environment, service interface should be unified and independent from particular platform and programming language.
- Lightweight communication protocol: Many kinds of embedded devices are getting connected to a smart space, and some may have quite limited capability. In order to support various devices, the resources required for communication should be as small as possible. Complex communication protocol may cause issues such

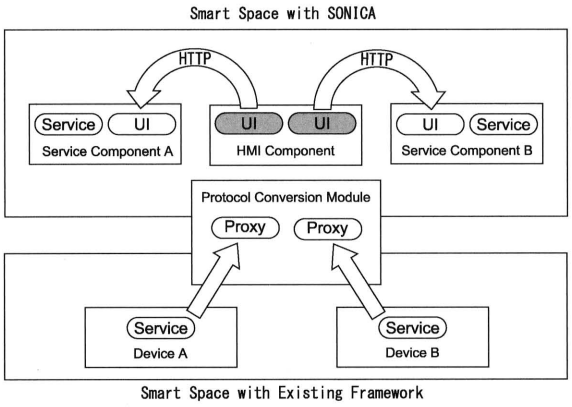

Fig. 1 Device Interaction.

as cost, power consumption, and increase in network traffic.

- Scalability: According to sophistication and spread of smart spaces, the number of connected devices should be increasing. Scalable interaction architecture is required for such environment.
- Flexible UI: To accomplish interactions among many kinds of devices, UIs required for operation should be simple. However, some high-end devices have maneuverable and user-friendly interfaces. Providing rich UIs by leveraging such interfaces depending on the feature and capability of devices improves usability.
- Support for existing framework: Some of existing smart spaces are self-contained and not suitable for collaboration with other smart spaces. In order to spread smart spaces, upcoming frameworks should have a mechanism to support such existing frameworks.

3. Design of SONICA

According to the requirements described in the 2nd section, we have designed SONICA (Service Oriented Network Interoperability for Component Adaptation) for control and management of embedded devices in smart spaces.

3.1 Components

A SONICA system is composed of service components (SC) and human machine interface components (HMIC). An SC works as an HTTP server and offers its functions as web services. According to a request from a user or other SCs, an SC provides its services. An HMIC is a component that works as an interface of users and SCs. Users can use services of an SC via an HMIC by operating the UI offered from the SC. A single physical device may contain one or more components.

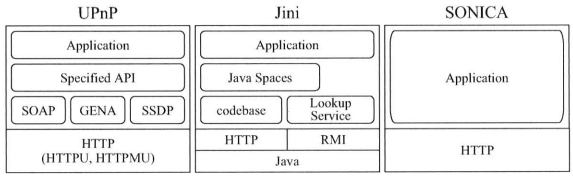

Fig. 2 Module Structure of UPnP, Jini and SONICA.

3.2 Device Interaction

Figure 1 shows the outline of device interaction on SONICA. Device interaction of SONICA is based on RESTful web services. As a RESTful web service employs HTTP methods as its service interface, it has high scalability, and can be implemented in many programming languages on various platforms. Consequently, SONICA is suitable for cross-platform environments, and can construct scalable smart spaces. As communication messages of SONICA for requesting services are encoded in x-www-form-urlencoded scheme, SONICA can support resource-limited devices which can't afford an XML parser. Such devices can produce XML documents by simple string operations without XML parsers, even if they are required to response in XML format.

For embedded devices, most of HTML implementations are based on HTML 4. And methods available for submitting a form data are only GET and POST on HTML 4. Therefore, this paper focuses on GET and POST.

A SC provides its UI to HMICs as well as services, and a HMIC just renders the UI. There is no need to prepare UIs of each service in advance. Therefore, a HMIC can adapt a lot of SCs and can support upcoming SCs without software updates. SONICA assumes that every HMIC has capability to render simple HTML UIs; however SCs can offer more maneuverable UIs depending on the feature and capability of HMIC.

Some popular smart spaces based on existing frameworks such as IEEE 1394 have no web communication interfaces. SONICA can support and collaborate with these smart spaces by adding a proxy component called protocol conversion module (PCM). A PCM deploys functions of devices on other smart spaces as RESTful web services. According to requests, a PCM interacts with the devices instead of other SONICA components.

Table 1 Evaluation Results of Each Framework.

Framework	UPnP	Jini	SONICA
Maximum Consumed Heap Memory [KB]	more than 3000	2367.5	935.8
Response Time (Average) [ms]	1693.9	16.9	10.2

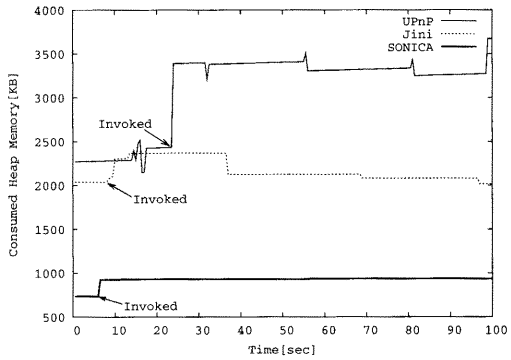

Fig. 3 Heap Memory Consumption.

4. Performance Evaluation

In this section, we evaluate the performance of UPnP, Jini and SONICA. **Figure 2** shows the module structure of each framework.

4.1 Implementation of Frameworks and Services

We implemented a simple service for evaluation with each framework: UPnP, Jini, and SONICA. The implemented service just returns the current time using java.util.Date class. To be fair, every framework and implemented service is written in pure Java, and evaluation is performed on J2SE 5.0.

For UPnP, we used CyberLink for Java[17] as its protocol stack. In this evaluation, we adopted Xerces as an XML parser for CyberLink, although KXML is also available.

We used Java RMI as a communication mechanism for Jini service. To exclude the resources for service discovery, we did not use a lookup service but rather the rmiregistry we describe later. We also used a local file system for codebase and not a HTTP server.

We designed and implemented a tiny HTTP server for SONICA SCs. The server adopts thread pooling model, and can handle HTTP GET and POST requests.

4.2 Evaluation Procedure and Results

We measure the consumed heap memory size and response time for UPnP, Jini, and SONICA. **Figure 3** shows variation in heap memory usage when each service is invoked. **Table 1** shows the maximum size of the consumed heap memory and the response time between sending the request and receiving a response. Usage of heap memory is measured every 500 ms for 100 seconds using jstat, which is a Java VM statistic data monitoring tool. Measurements are taken in two situations: when the service is waiting and running. The values shown as the usage of heap memory are sums that are delivered by multiplying the consuming rate by the reserved size in the following three generations: NEW Generation (includes the Eden Range and Survivor Range); OLD Generation; and Permanent Generation. The value shown as Jini includes not only resources for the implemented service itself but also for the rmiregistry. The service discovery mechanism is implemented in quite different ways on each framework, for example SSDP (Simple Service Discovery Protocol) on UPnP and the Lookup Service on Jini. To make a fair comparison, the resources used for service discovery are not included in this evaluation.

Compared with the other two frameworks, UPnP has a 1693.9 ms response time. This is attributed to the cost of parsing each communication message described in XML. UPnP also consumes more than 3000KB of heap memory for the XML parser. Jini works well in the area response time, but on the other hand, it needs much more memory for the rmiregistry. In the RMI communication sequences, marshalled objects are transferred so the network traffic is thought to be heavier. On the first request, marshalled objects are transferred to the client. However, on consecutive accesses, the cached service works in the client VM. So the Jini response time changes whether it is the first request or not. In this evaluation, we used rmiregistry and a local file system for the codebase as Jini implementation. However, on a real Jini system with a lookup service and a http codebase, many more resources are needed, and the response time is thought to be longer. Compared with UPnP and Jini, the service with SONICA responds rapidly, and the consumed memory size is less than 1MB. These results lead us to the conclusion that SONICA is more suitable for embedded devices with limited resources.

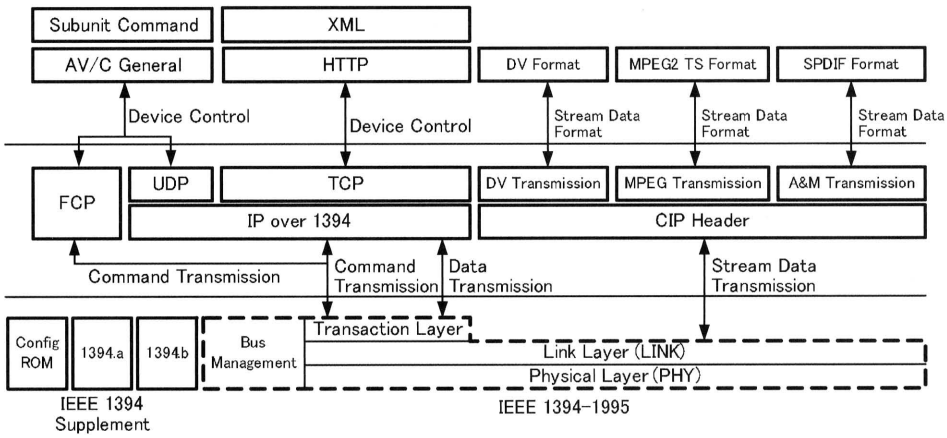

Fig. 4 SONICA Protocol Stack over IEEE 1394 with IPover1394.

Table 2 Implementation Environment.

CPU	Intel Pentium 4 (3GHz)
OS	Linux 2.6.9 (customized for IPover1394)
Java	Sun JDK 5.0
IEEE 1394 Interface	Buffalo IFC-ILP4
Video Camera	Apple iSight

5. Evaluation through Application

To verify that SONICA can support and collaborate with existing framework, we implemented a video camera controlling system on IEEE 1394 network. The implementation environment is shown in **Table 2**.

5.1 SONICA over IEEE 1394

In this study, we implemented SONICA over IEEE 1394 (a.k.a. iLink or FireWire)[18)−20)]. IEEE 1394 is a serial interface that supports up to 800Mbps data transmission, and has an isochronous transmission mode for real-time streaming data, as well as an asynchronous transmission mode. **Figure 4** shows the protocol stack for this implementation. In this system, we employ IPover1394[21)] protocol to constructs an IP network over IEEE 1394. IPover1394 uses asynchronous transmission mode for transferring IP packets.

5.2 Camera Application with SONICA

Figure 5 shows an overview of the communication procedure within SONICA network. The system consists of 4 devices: HMIC Device, Content Manager as SC, IIDC PCM as PCM to control the IEEE 1394 video camera, and Apple iSight. In this system, Content Manager offers a service to save images sent from other devices. IIDC PCM works as a proxy that deploys functions of IEEE 1394 digital cameras as RESTful web services, and interacts with cameras instead of other SONICA components.

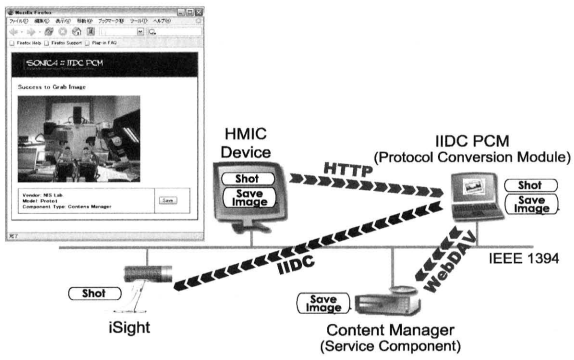

Fig. 5 Communication Procedure.

In order to use the IEEE 1394 digital camera, we developed a library named libdc1394j that controls the library called libdc1394 from Java. On Linux, libdc1394 is available as an IIDC control library. Libdc1394 is described in C Language and is impossible to use directly in Java. To use the library directly in Java, we implemented a shared library that includes a controlling mechanism that is described in the native language, and a stub that loads this library from Java using JNI (Java native interface). This stab can be used as if it is the Java library. **Figure 6** shows a libdc1394j software stack. Libdc1394j contains a stub which is an interface of libdc1394 and extra libraries. The extra libraries include the image format translation method implemented in the native language. In libdc1394, error handling is strongly dependent on applications; however, it is difficult to manage errors that occur in the native methods. Thus, libdc1394j manages errors in the native methods, and then informs the Java applications. Call by reference is heavily

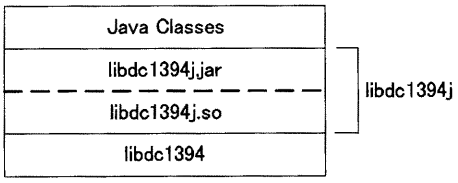

Fig. 6 Libdc1394j Software Stack.

used in the libdc1394 original source code; however JNI does not define passing method for pointer variables. In the Java application, we handle the pointer variables as integer variables.

The working sequence of this system is as follows: (1) A user accesses to the UI of IIDC PCM via HMIC Device. Available services of 1394 digital cameras are shown in the UI. (2) The user selects 'Shot' service on iSight. Using libdc1394j, IIDC PCM controls iSight to take a picture. The captured picture is automatically transferred to IIDC PCM. (3) The picture is shown in HMIC Device. 'Save Image' service on Contents Manager also shown in HMIC device. (4) When the user selects 'Save Image' service, IIDC PCM invokes the service on Content Manager. The picture is transferred to Contents Manager using WebDAV protocol. (5) The user accesses the UI of Content Manager, and confirms that the transferred image has been saved correctly.

6. Conclusion

In this paper, we described requirements on frameworks for smart spaces with embedded devices, and then proposed a new framework based on the requirements. The proposed framework is designed to have high scalability and be suitable for cross-platform environments by employing RESTful web service for device interaction architecture. We measured response time and memory consumption for UPnP, Jini and SONICA, and SONICA responded more quickly and consumed less memory than the other technologies. To verify that SONICA can support and collaborate with existing framework, we implemented practical system over IEEE 1394 network to control a IEEE 1394 digital camera through the protocol conversion module.

Acknowledgement

This work is partially supported by research fellowship for young scientists (No. 20-04285) from Japan Society for the Promotion of Science (JSPS) in 2008-2010 and Grants-in-Aid for Scientific Research (B) (No. 18300021) from Japan Society for the Promotion of Science (JSPS) in 2007-2010.

References

1) DLNA (Digital Living Network Alliance), "DLNA Overview and Vision White Paper 2007", 2007. http://www.dlna.org/news/DLNA_white_paper.pdf.
2) Sun Microsystems, Jini Network Technology. http://www.sun.com/software/jini/.
3) Jini.org, "Jini Architecture Specification", 2006. http://www.jini.org/wiki/Jini_Architecture_Specification.
4) Home Audio Video Interoperability (HAVi) Organization, "White Paper, HAVi, the A/V digital network revolution". http://www.havi.org/pdf/white.pdf.
5) ECHONET CONSORTIUM. http://www.echonet.gr.jp/.
6) R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D dissertation, University of California, Irvine, 2000.
7) UPnP Forum, "UPnP Device Architecture Version 1.0", 2006. http://www.upnp.org/specs/arch/UPnP-DeviceArchitecture-v1.0.pdf.
8) W3C Recommendation, "SOAP Version 1.2 Part 0: Primer", 2003. http://www.w3.org/TR/soap12-part0/.
9) W3C Recommendation, "SOAP Version 1.2 Part 1: Messaging Framework", 2003. http://www.w3.org/TR/soap12-part1/.
10) W3C Recommendation, "SOAP Version 1.2 Part 2: Adjuncts", 2003. http://www.w3.org/TR/soap12-part2/.
11) W3C Recommendation, "SOAP Version 1.2 Specification Assertions and Test Collection", 2003. http://www.w3.org/TR/2003/REC-soap12-testcollection-20030624/.
12) J. Newmarch, "A RESTful Approach: Clean UPnP without SOAP," *Proc. the IEEE 2nd Consumer Communications and Networking Conference*, pp. 1–5, 2005.
13) J. Vayssiere, "Transparent Dissemination of Adapters in Jini," *Proc. 3rd International Symposium on Distributed Objects and Applications*, pp. 95–104, 2001.
14) K. Kadowaki, T. Koita, K. Sato, and H. Hayakawa, "Design and Implementation of Adaptive Jini System to Support Undefined Services," *Proc. 6th Annual Conference on Communication Networks and Services Research (CNSR2008)*, pp.577–583, 2008.
15) A. Kaminsky, "JiniME: Jini connection technology for mobile device," Technical report, Information Technology Laboratory Rochester Institute of Technology, 2000.
16) C. Lee, S. Helal, and D. Nordstedt, "The μJini Proxy Architecture for Impromptu Mobile Services," *Proc. the International Symposium on Applications and the Internet Worrkshop*, pp. 113–116, 2005.
17) Satoshi Konno, CyberLink for Java. http://www.cybergarage.org/net/upnp/java/index.html.
18) IEEE, "Standard for High Performance Serial Bus," IEEE Std 1394-1995, 1995.
19) IEEE, "Standard for High Performance Serial Bus Amendment 1," IEEE std 1394a-2000, 2000.
20) IEEE, "Standard for High Performance Serial Bus Amendment 2," IEEE std 1394b, 2002.
21) Internet Engineering Task Force (IETF), "IPv4 over IEEE 1394," Request for Comments (RFC) 2734, 1999. http://www.ietf.org/rfc/rfc2734.txt.