

Implementation and validation of UPnP for embedded systems in a home networking environment

T. Tranmanh, L.M.G. Feijs, J.J. Lukkien

Eindhoven University of Technology

Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Email: t.tran@tue.nl, l.m.g.feijs@tue.nl, j.j.lukkien@tue.nl

Abstract

Devices in a home environment are often equipped with general purpose network connections. There is an increasingly strong requirement that these devices cooperate in an autonomous fashion by using the functionality they find on the network. In the context of home networking several standard technologies have been proposed for this purpose of which Universal Plug and Play (UPnP) is an example. This article presents a way to implement a UPnP Application Programming Interface (API) to build UPnP applications on top of it. We also give a validation of the chosen architectural model and of the positive and negative points of UPnP in applying it for the given context.

Keywords: *home networking, embedded system, UPnP, service discovery.*

1 Introduction

Nowadays, devices in a home environment are often equipped with general purpose network connections. There is an increasingly strong requirement that these devices cooperate in an autonomous fashion by using the services (i.e., functionality) they find on the network. To that end devices need to be able to find out about these services, to find the devices that supply them to employ the services and, conversely, they must be able to advertise their own services. In the context of home networking several standard technologies have been proposed for this purpose of which Universal Plug and Play (UPnP) is an example.

The home environment is particularly difficult from a networking perspective. One has to deal with heterogeneous networks and with devices that come and go all the time; the resources are scarce and there can be absolutely no user intervention for configuration. This requires technologies that can effectively and efficiently accommodate change and complexity. UPnP [1], put forward by Microsoft in 1999, was designed for this. UPnP relies on the Internet Protocol (IP) to deal with heterogeneity. On top of that it uses standard Internet protocols enabling it to seamlessly fit into existing networks. Although it is rather

new and there are still few UPnP-enabled products on the market it is currently considered by many companies that are active in home networking environments.

At the Eindhoven Embedded Systems Institute (EESI) we investigate the value of UPnP for embedded systems in the home and corporate environment by developing an API for it and constructing an experimental environment. With this API we validate UPnP and we also investigate how well it combines with other technologies, such as Jini [3] and HAVi [4]. Since the UPnP forum only provides a UPnP specification, the way to implement an API is heavily vendor dependent. This article presents our UPnP API developed for the context of embedded systems.

The article is organized as follows. In section 2 we present an overview of UPnP including examples of how it operates. Section 3 contains the architecture of the API and some details about the implementation. In section 4 two prototypes are described as well as evaluation results of their use. Finally we give a conclusion about UPnP, including positive and negative points. We propose several alternative solutions and future research.

2 UPnP overview

UPnP is meant to be an architecture for peer-to-peer connectivity of devices of all form factors, ranging from PCs to rather simple appliances, within a limited physical environment (e.g., home, office or a public space). The envisaged cooperation is through the roles of a *control point* and a *device*. A device is a container of *services*, where a service is the smallest unit of control in a UPnP network. A service exposes actions and state variables. A control point manipulates the device through the services. These three elements are the basic building blocks of a UPnP network. We use the term *terminal* to refer to a control point or a device. In fact, a terminal may play both roles. There are six issues involved in UPnP operation. They are addressed using Internet standards.

Addressing: When the control point or the device connects to the network it must obtain an IP address. If available, the Dynamic Host Configuration Protocol (DHCP) is used for that purpose. If not, Auto-IP is an alternative. In this way

resource-scarce networks are included for UPnP operation.

Discovery and advertisement: The control point needs to find devices of interest; the device needs to advertise itself. The Simple Service Discovery Protocol (SSDP) [5] is used here. Again this allows configuration-free cooperation.

Description: The control point learns about the device and its capabilities (its services); the device needs to specify these. This description is an XML document for the device and an XML document per service that can be retrieved from the device.

Control: The control point invokes actions from the device. The Simple Object Access Protocol (SOAP) [7] is used.

Eventing: The control point subscribes to events and then listens to the state changes of the device. The General Event Notification ArchitectureBase (GENA) [6] is used here.

Presentation: The control point controls the device and/or views the device status using a Web browser.

In a sense one can say that UPnP combines several disparate protocols into a single architecture. Using this, a terminal can dynamically join a network, obtain an IP address, convey capabilities, and learn about the presence and capabilities of other terminals. Terminals can subsequently communicate with each other directly, assuming control point and device roles for services.

The example of a TV set helps to explain this. We refer to it as *TVDevice* and it admits the following operations: *Switch TVDevice on/off*, *Change channel*, *Change volume*, *Change color* and *Change contrast*. To define *TVDevice*, we divide this functionality into two services: the first one, named *ControlService*, fulfils the first three functions and the second one, *ScreenService*, fulfils the last two. Three XML files are required to specify *TVDevice*: one file specifies *TVDevice* and two others specify its two services. Now, a cooperation of *TVDevice* with some control point *TVControl* proceeds as follows.

- When *TVDevice* connects to the network it obtains an IP address and advertises itself through SSDP.
- When *TVControl* connects to the network it obtains an IP address and issues a discovery request through SSDP.
- Subsequently, *TVControl* retrieves the device description and gets a list of associated services, it retrieves service descriptions of interesting services, it invokes actions to control the service and it subscribes to the service's event sources. Each time the state of the service changes, an event to the control point will be generated.

Retrieving XML documents is done through HTTP; in addition, SSDP, SOAP and GENA use HTTP as carrier. As a result the architecture contains HTTP as a layer and several HTTP servers must be present, one of which is a web server.

Different categories of UPnP devices will be associated with different sets of services. Consequently, different working groups will standardize on the set of services that a particular device type will provide. All of this information

is captured in the XML device description document that the device must host.

UPnP uses open standard protocols (TCP/IP, HTTP, XML). However, other technology could be used to network devices together for reasons of cost, technology requirements, or legacy support. Examples are Jini and HAVi. These can participate in the UPnP network through a UPnP bridge. This bridge acts as a proxy application between UPnP terminals (devices, control points) and components which belong to other networking technologies. Discussing these other technologies is beyond the scope of the current paper.

For a complete description of UPnP we refer to the site of the UPnP forum [2].

3 Design decisions

The API

Two essential components of a UPnP-based system are the control point and the device. We have constructed an API for both and implemented it in Java. The architectural model is described in Figure 1. There are 5 layers in this model. The top layer is the application which is built on top of the API - the second layer. The API is a collection of several classes and the application is built from these classes. The UML [10] class diagrams for the API of the control point and the device are given in Figure 2. The specific device application is an extension of class *Device*; the specific control point application extends class *Discovery*. Within the control point, classes *CDevice* and *CService* represent proxies of the actual device and service. Class *GUIControl* represents the framework for the user interface if needed. Layer 3 includes three components that implement the three protocols SSDP, SOAP and GENA. Components in layer 3 are started by the API. The dotted line in Figure 1 represents a call-back function; the direction of the arrow represents the direction of data transfer.

Layer 4 represents two known protocols: UDP and TCP. To implement the discovery, the SSDP component

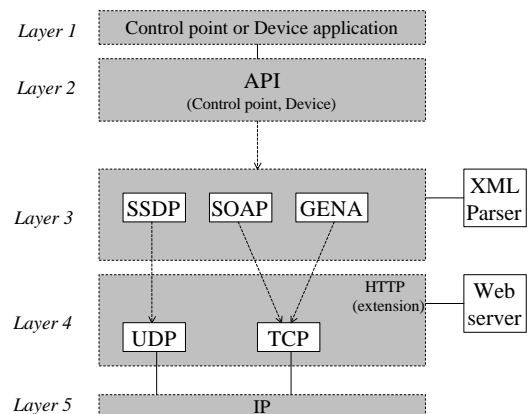


Figure 1: architectural model of the system

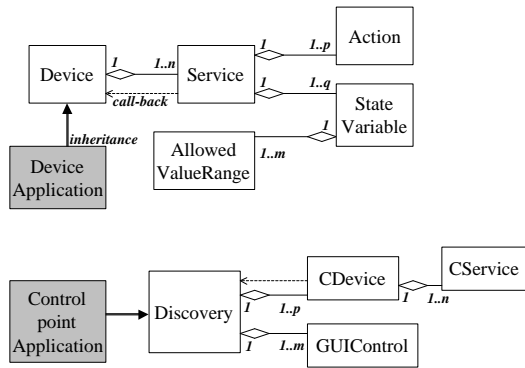


Figure 2: class diagrams for the API

uses UDP. For controlling (SOAP) and eventing (GENA), TCP is employed. The format of data transfer between layer 3 and layer 4 conforms to the HTTPe protocol which is an extended version of HTTP/1.1 with the purpose of fulfilling the requirement of SSDP, GENA and SOAP. In layer 4 two essential classes are *UDPListener* and *TCPListener*. *UDPListener* is an HTTPe server running on top of UDP to handle the service discovery process in SSDP. *TCPListener* is an HTTPe server running on top of TCP to deal with the requirements of SOAP and GENA.

Since IP is the last layer, every element in the system must be IP-enabled.

There are two components that do not belong to any layer. The component named *XML Parser* can be invoked by any component in layer 3 to parse XML content. The component named *Web server* is an HTTP/1.1 server running only in the device. This server handles the requests for the device and service descriptions.

To explain the system in more detail, we show briefly how SSDP, SOAP and GENA work. When a device joins a network, it advertises itself to the network with a certain frequency. We call this *advertisement*. When a control point joins a network, it searches for the device that it can control. We call this *discovery*. The SSDP protocol includes both Advertisement and Discovery. Module SSDP

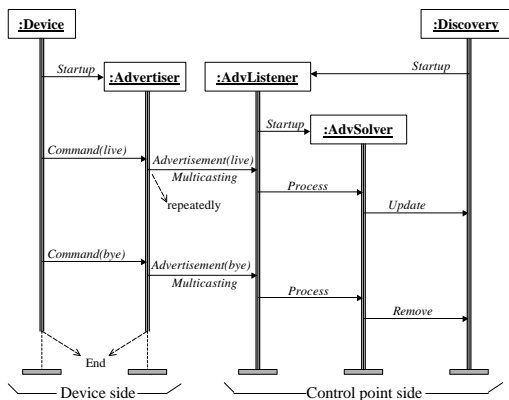


Figure 3: the advertisement

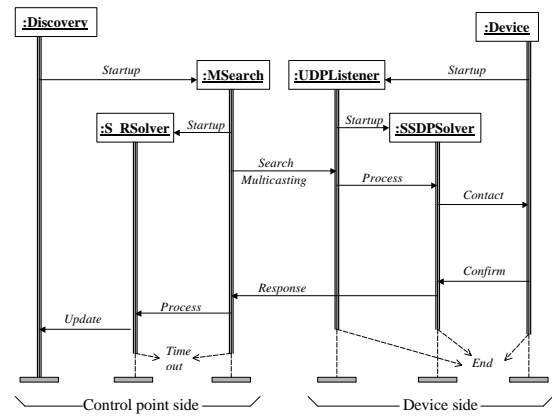


Figure 4: the discovery

includes the following classes:

- **AdvListener:** an HTTPe server running in the control point to handle the Advertisement from devices.
- **AdvSolver:** processes the data given by AdvListener.
- **MSearch:** initiates the discovery from the control point side.
- **S_RSolver:** handles the responses from devices when the control point processes the discovery.
- **Advertiser:** initiates the advertisement from the device side.
- **SSDPSolver:** processes the data given by UDPListener.

The UML sequence diagrams for advertisement and discovery are shown in Figures 3 and 4.

Component SOAP includes only the class named *SOAP-Solver* running in the device to handle the control commands issued by the control point. The UML sequence diagram for the control process is shown in Figure 5.

Component GENA implements the eventing process. It includes the classes:

- **SubsSolver:** runs in the device to handle the subscription requests from the control point.

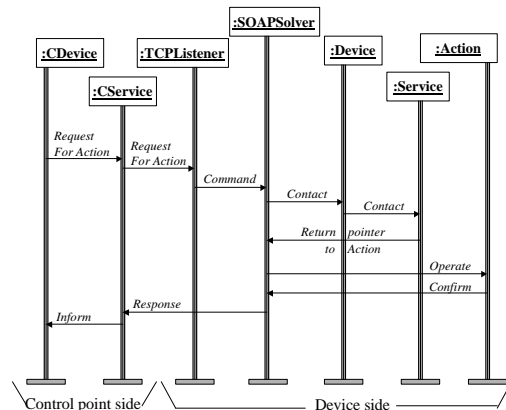


Figure 5: the Control process

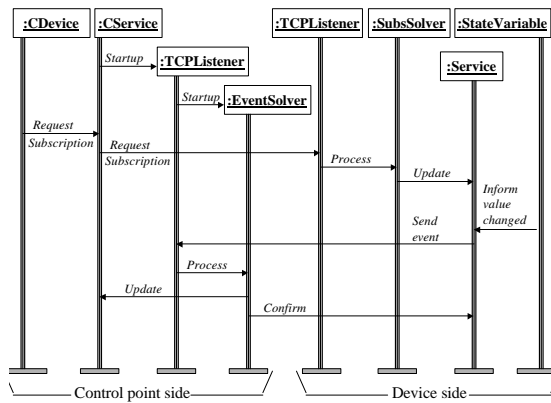


Figure 6: the Eventing process

- **EventSolver**: runs in the control point to handle the events sent by the device.

The UML sequence diagram for the eventing activity is shown in Figure 6.

System in operation

Figure 7 depicts an example of a device in operation. This particular device has three services where services 2 and 3 are mapped onto the same port. Within a service we find the actions they provide. Associated with each port is a server task that dispatches messages from the control points to the service for which they are intended. Besides these there is another server to deal with incoming UDP messages for the service discovery. All servers in Figures 7 and 8 use HTTPe, except the Web server.

The implementation of the services amounts to executing the actions that establish the required physical effect upon receipt of the relevant control commands. The relation with the actual device can be implemented in a number of ways, e.g. through a direct mapping onto the hardware or by using a device proxy which communicates with the actual device through some proprietary protocol. In this way it is possible to include different protocols and to let UPnP have bridge functionality at the functional level. For

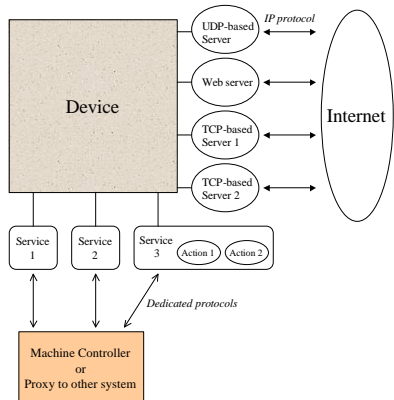


Figure 7: an overview of a device in operation

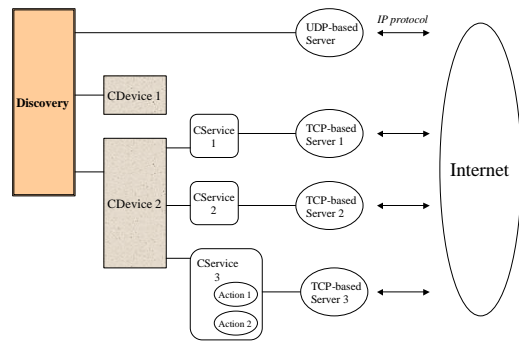


Figure 8: an overview of a control point in operation

example, a HAVi-enabled VCR can be controlled in this way.

Figure 8 presents the control point in operation. Within it we find proxies for devices: *CDevice1* and *CDevice2*. Within the device we find proxies for the services. Again, each port has an associated server, in this case to deal with event notifications. *CDevice2* is controlling the device from Figure 7.

Observe that at both sides (control point and device) several HTTPe servers must be running to handle the discovery and peer-to-peer processes. In view of the limited processing power of embedded systems the number of concurrent tasks should be limited. We propose a model for a HTTPe server as shown in Figure 9. A HTTPe server maintains a FIFO queue for all client requests. So, per port, every request is handled sequentially. While we choose this, there are two aspects to take into account.

- each request-response must be handled in a short enough time t as prescribed by the UPnP standard,
- the queue sizes must be long enough so that the queue can contain all requests.

These aspects are closely related: a longer t leads to a larger value of s .

The complete explanation of our design and implementation is found in [8]. The code size of the device API is 80.1 Kbyte and of the control point API 106 Kbyte. When they are compiled into Java byte code, the device API counts for 62.3 Kbyte and the control point API counts for 85.7 Kbyte.

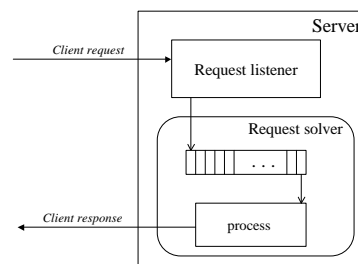


Figure 9: HTTPe server model

4 Experiments

In order to validate and evaluate the API we have built two prototypes. The validation concerns various aspects of UPnP:

1. How easy is it to build an application?
2. What is the usability from a users perspective?
3. Does the protocol scale to tens or hundreds of devices and control points?
4. Are the resource requirements reasonable?

The first prototype is *TVControl* controlling *TVDevice* that we discussed before. This prototype is a simulation of a user interface running in a computer to control the television using the two services *ControlService* and *ScreenService*. A system consisting of a single device and control point was easily constructed. The services were described in XML files in such a way that a user interface could be generated automatically. In order to search a device the user has to enter the identification number or the type of the device. This is a disadvantage as it prohibits a directory lookup.

Next, we increased the number of *TVControl(s)* gradually from 10 to 100. From a functional perspective the system still works well but the time lapse between request and response becomes unacceptable, especially in the eventing part: the 101st *TVControl* receives the event after roughly 10 seconds. The reason for this problem lies mainly in the use of HTTP as the transport protocol. If we have 101 *TVControl(s)*, then the 101st *TVControl* receives the event only when the previous 100 *TVControl(s)* have already received it, i.e., after setting up and destroying 100 TCP channels. Although eventing is, in fact, multicasting this is not implemented effectively. The result could be slightly better by communicating the events concurrently but that would impose a high demand on the processing power of the embedded device. Clearly, the protocol is not intended for that.

We see that the use of HTTP has both advantages and disadvantages. On the upside is the standardization which allows communication of a device with a Web browser. In addition, HTTP is connection-less which is good in the dynamic context of devices that come and go all the time. And finally, an HTTP communication itself is realized using a connection-oriented protocol which guarantees reliability. On the downside there is the significant overhead in setting up connections, the relative complexity of the protocol implementation and a significant timeout penalty when a device disappears *during* an HTTP transaction.

Another useful application of *TVDevice* is obtained if the screen *itself* becomes available to the control point. This would mean that a real-time stream is sent to the device, which it subsequently displays. UPnP lacks support for this at two places. First, this cannot be specified within the XML specification; second, communications within the UPnP framework are via HTTP which is not amenable to real-time streaming. It might be possible to use UPnP as

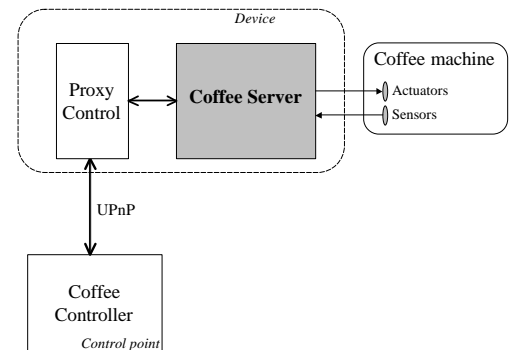


Figure 10: UPnP-enabled coffee machine

a framework to setup the real-time stream as, e.g., in the Session Initiation Protocol (SIP). However, this would not be an integrated control of the device anymore.

The second prototype is a coffee machine controller. We have built a Web-enabled coffee maker before, for the purpose of research on Web-enabled embedded systems[9]. The user can control a real coffee machine via a Web browser using an applet and a dedicated protocol. There is an application server that mediates between the Internet connection and the coffee machine, named *Coffee Server*. From a Web browser, the user can do the following: *Switch machine on/off*, *Select taste level*, *Inspect amount of coffee in the container* and *Inspect temperature*. In order to bring the system to work in a UPnP environment, we built a control point, named *CoffeeController* and a proxy, named *ProxyControl* (see Figure 10). *CoffeeController* also can process the four commands. This prototype shows how to upgrade a legacy and non-UPnP system to become UPnP-enabled.

5 Conclusion

We have built a UPnP API and implementation. Two aspects that drive the design process are that the API is built for embedded systems and that UPnP is applied for home networking environments. We also have built two prototypes for the purpose of evaluation. It is clear that UPnP can really be used for embedded systems in the home environment provided that IP is available. In applying UPnP for this kind of environment, several conclusions can be drawn:

Flexibility of design: Even though the UPnP forum has given the UPnP specification, vendors are free to construct their own environment, depending on the type of in-house devices they have. In particular, the API and its implementation are not defined. In our case, the API is designed for embedded systems with limited processing power.

It is not too difficult to upgrade a non-UPnP device to become UPnP enabled. However, for a system with several standards living together like UPnP, Jini and HAVi, it is not easy to solve the problem of interoperability.

Eventing mechanism: UPnP employs TCP to transport

events. If the system has a limited number of control points and when there are few events this is suitable, since TCP maintains the reliability. Within a system with hundreds of control points (most of them just playing the role of monitoring the device), the use of TCP can cause the system to break down due to too much traffic. In addition, in case a control point is removed from the system the TCP timeout mechanism causes long delays. It appears that a new UDP-based protocol should be designed that deals more effectively with the broadcasting and dynamic connections.

User interface: The way of searching for a device can be inconvenient for a user. To find a device, one has to type either its type, its name or its identification number. In practice, a user likes to find his device by using some *browse-and-click* way. For example, to find the television, one may choose the first floor of the house, then come to the living room and click on the television. One should not have to remember an identification of the television. In general, how to create a suitable naming mechanism in UPnP is still an open question. The above example could be included as just another UPnP service though.

Quality of service/content: The way to describe a device by using XML also raises another question: is an XML-based document rich enough to cover such cases as real-time requirements or very complex interfaces? Research in this direction can lead to a specification of the quality of service (QoS) and the quality of content (QoC) in the home networking environment through XML documents.

Extensibility: Moving from a single environment to wider area networks introduces several new challenges. One UPnP model may no longer suffice. Consider the case when the user wants to control the house remotely, from his office. The existence of a UPnP gateway (or bridge) is a reasonable answer. Then, how about Internet firewalls, and security concerns in general. These things have not been addressed in UPnP explicitly. The security concern in UPnP seems to be delegated to the problem of maintaining the security in each HTTP server in the UPnP architecture.

References

- [1] *Universal Plug and Play specification v1.0*. Online: <http://www.upnp.org/resources/documents.asp>.
- [2] The Universal Plug and Play Forum. Web site: <http://www.upnp.org>.
- [3] K. Arnold et al. *The Jini Specification*. Addison-Wesley Longman, Reading, Mass, 1999.
- [4] *HAVi specification 1.0*. The HAVi Organization, January 2000.
- [5] Yaron Y. Goland et al. *Simple Service Discovery Protocol/1.0*. Internet Draft, October 1999.
- [6] J. Cohen et al. *General Event Notification ArchitectureBase: Client to Arbitrator*. Internet Draft, September 2000.

- [7] *Simple Object Access Protocol(SOAP) 1.1*. World Wide Web Consortium, May 2000.
- [8] T. Tranmanh, P.J.F. Peters, J.J. Lukkien, L.M.G. Feijs. *Design and implementation of an API for UPnP-based embedded systems*. Eindhoven Embedded Systems Institute, Eindhoven University of Technology, 2002.
- [9] P.J.F. Peters J.J. Lukkien, M.F.A. Manders and L.M.G. Feijs. *An architecture for web-enabled devices*. In *Proceedings of the 2001 International Conference on Internet Computing, Las Vegas*, June 2001.
- [10] J. Rumbaugh, I. Jacobson, G. Booch *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, Reading, Mass, 1999.