ANALYSIS AND IMPROVEMENTS OF THE EVENTING PROTOCOL FOR UNIVERSAL PLUG AND PLAY

Y. Mazuryk
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email: y.mazuryk@tue.nl

J. J. Lukkien
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email: j.j.lukkien@tue.nl

ABSTRACT
UPnP is a widely-spread connectivity standard, which allows networked devices to cooperate in an autonomous fashion by using functionality found on the network. In this article we validate UPnP as a service-oriented architecture. We identify shortcomings of the standard and propose solutions. In our view, eventing is the weakest mechanism in UPnP technology. We propose extensions to the existing eventing protocol in UPnP, which allow overcoming identified problems. We compare our solution with standard UPnP with respect to performance.

KEY WORDS
Protocol, eventing, UPnP, home networking

1 Introduction

The Service-Oriented approach is a rather new paradigm for distributed computing which can, in fact, be viewed as a supplement of the object oriented programming paradigm. [1]. However, while object oriented programming focuses on how the software components are constructed, service oriented programming focuses on what components can do. In addition, service oriented programming is aimed at a networked context, building applications from collaborative services [2].

A service in a service oriented architecture represents a contractually specified functionality. The service is made available at service access points through an interface. This interface includes actions and state variables. An example of a service is the delivery of messages which is accessed at the transport layer of a network stack through some interface and in which the state is represented by the connection between source and destination. In contrast, service oriented architectures focus on the network as access point for the service. There are two roles: the service provider and the service user. They are similar to the notions of client and server except that they don't represent an architectural choice but roles that can be played by one and the same application. A service-oriented application in a networked context generally has four dependencies.

1. It may serve a certain end-usage, e.g., interfacing with an end user or system;
2. It may expose services on the network. These services represent functionality that can be used by other applications;

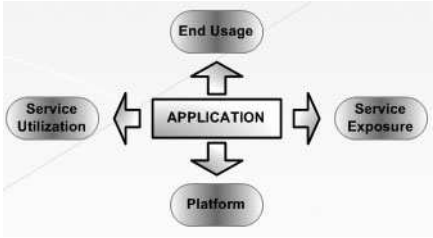

Figure 1. Service Oriented Application - Context

3. It may use services it finds on the network, either to support the services it exposes or to serve the end-usage;
4. It depends on a platform (OS, middleware).

The last dependency is there for all applications. The other three do not need to be present in every application. For example, regular, single processor applications only have the first dependency. Many common applications use both the first and the third dependencies. Current applications that expose services usually are special purpose and dedicated, like web servers or DNS servers. With the advent of service oriented architectures this situation may change in that exposing services becomes more common, as well as building new applications from these.

Corresponding to this view on building applications, the key aspects of service-oriented architectures [3] are:

- *discoverability*: a service user has to be able to discover the appropriate service in the system. A service (provider) has to inform the system that it "exists" and is able to handle requests from service users;
- *location transparency*: user and provider are not bound to a certain network node, and can roam the network;
- *loose coupling (late binding)*: the binding between user and service is performed at runtime. A client doesn't have any prior knowledge about a specific service before it is being discovered. Knowledge about location, identity and history is kept minimal and the service user is able to deal with sudden loss of the service;
- *interoperability*: the ability of applications to use each other's services regardless of programming language, Operating System or other implementation specific issues;
- *conjunctivity*: the ability to use or combine services in ways not conceived by their designers.

There are four key mechanisms which support the above properties:

- *description*: the identification of the service interface, which is a contractual agreement between the service user and the service;
- *advertisement & discovery*: services make themselves known to service users and vice versa;
- *control*: the mechanism used by the service user to request certain actions from a service;
- *eventing*: the mechanism used by the service to reach its current users.

In this paper we evaluate an example of a service oriented architecture, viz., Universal Plug and Play (UPnP) in terms of its usability and performance, and also to what extent the above general properties are supported. We do this by using the simple example of a file access service implemented using UPnP technology. The service should fulfill the following requirements:

- the list of accessible files is changing over time;
- users can independently monitor the state of a single file and receive notifications when the file is edited or removed;
- actions to be exposed by the service address creation, removal, reading and writing files.

The paper is structured as follows. In section 2 we discuss the UPnP architecture, point out weak points with respect to the chosen example and propose solutions. In section 3 an improved eventing protocol is described. Then we analyze the performance of the proposed protocol in section 4. We conclude with issues that require further investigation.

2 UPnP

2.1 General Remarks

The UPnP standard is rapidly growing in popularity, while at the same time the standard is still evolving. We refer to [4] for a description. Summarized rather crudely, UPnP services, contained in *devices* expose observable state variables and actions that can subsequently be accessed from so-called *control points*. Devices can report changes in the state variables to control points through *events*. The specification ("contract") of this interface is statically determined and can be retrieved from the device at runtime.

The primary focus of UPnP is the control of certain functionality from a remote location. The participants are not equal in rights and responsibilities as well as the boundary between the controller and controlled object are clear. Regarding the taxonomy of networked devices in [5], a UPnP-enabled device would classify as a network-central device. Indeed, the only thing it provides is remote control of its functionality and, therefore, a UPnP-device is more a system by itself than a part of a large distributed system that it uses for its functionality. Nevertheless, controlling a UPnP-device is rather straightforward, and does not require much additional configuration on the controller. Also, UPnP can be used in a more general way than it was designed for.

A specification of our file access service in reads as in Figure 2. This specification is not pure UPnP. Obtaining

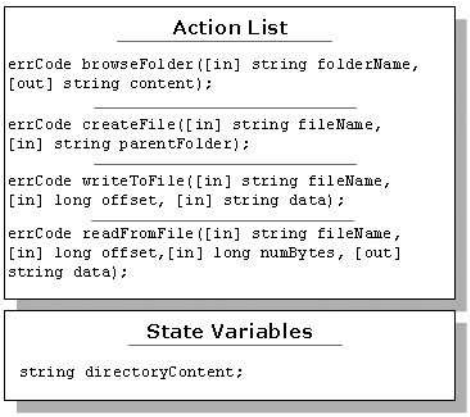

Figure 2. File Access Service Specification

UPnP specification for such a service is not straight forward and cumbersome, as we will show later. The specification of a service is retrieved from the device in two steps: first, the device is found, second, the services within the device.

2.2 Discovery

UPnP makes use of Simple Service Discovery Protocol [6] in order to discover network devices and services which are provided by them. Although the protocol is rather simple, it has a number of disadvantages. Network devices periodically advertise their presence by multicasting. Multicasting is used also for discovery requests. One can imagine, that such an approach is not scalable. In case the network contains a high number of UPnP-enabled devices, multicast traffic will be significant. Alternative approaches use mediated discovery, in which client interacts with the information brokers in order to retrieve service information [7], [8] and [9]. A comparison of various discovery mechanisms is presented in [10].

2.3 Actions

Actions in UPnP are implemented through the SOAP protocol; messages are transported via HTTP and are handled by an embedded web server. A complete transaction consists of such a call followed by a reply. In the UPnP specification it is recommended that processing an action request by the UPnP service should not take more than 30 seconds. This is due to the fact that interaction between client and service is performed via the HTTP protocol which employs a "request-reply" mechanism with a timeout at the requesting side. During the whole period a TCP connection is open. In view of the requirement of loose coupling we consider this an unfortunate choice. First, a fixed timeout does not serve all uses. For example, in the file service, a large amount of data may be transferred as a reply to an action call, e.g., while reading a large file. Conversely, actions that require rapid feedback may need a much smaller timeout. Second, the connection oriented TCP carrier takes a relatively long time to establish while it is only used for a single exchange. It also introduces its own peculiarities when the connection is broken. Rather than having a large timeout, the request and response can be separated the response being a call-back. REPHRASE, make

a remark that it was not done in this paper. For this, a carrier protocol is needed that is reliable but not connection oriented, since UPnP should support low-latency interaction.

2.4 Arguments

In UPnP, every argument of an action has to be bound to some state variable which limits the design freedom substantially. Either "fake" state-variables have to be introduced by the designer, or the actions should be designed in such a way that every argument indeed is related to a state variable. For example, in the file-write action of the file-access service a filename and data buffer are passed as an argument. However it is not possible to relate those to the state variables of the service, which are statically determined in the service advertisement. This is why the spec in Fig. 2 is not a valid UPnP specs.

2.5 Events

The events in UPnP are signals about changes in values of state variables. Through the event notification the subscriber receives the new value of the state variable. This again limits the flexibility of the UPnP eventing system as events should be separated from state variables. For example, the file-access service could inform the users whenever a file was modified. Having 1 state variable for every file is not effective as it would imply changing XML descriptions of the service upon adding and deleting files.

Another issue which limits the usability of eventing even more, is the fact that clients cannot subscribe to individual event-types but only to all events from a service. In this way a lot of unnecessary traffic is generated and clients receive messages that they simply ignore. Thirdly, the broadcasting nature of eventing also leads to inefficiency when mapped directly onto a connection oriented protocol[11].

The eventing protocol is GENA [12], transported by TCP. A complete transaction consists of an event delivery to a subscriber followed by a reply by the receiver. As in the case of actions, this brings additional limitations. For every event notification a TCP connection is setup and destroyed which limits event rates to be in the per-second range. In addition, the high mobility of networked devices can have a significant influence on the event notification delivery - the control points may leave the network while their subscriptions to events are still valid, causing excessive TCP timeouts.

2.6 The API and Anonymity

Since UPnP only defines the protocol and not the API there is quite some freedom in deciding which information is passed to the application. In most UPnP SDKs it is not clear at the API level where the control action comes from. The service application has to go down to SOAP messages to determine the source of the control action. This anonymity at the API level makes implementation of session-based services consisting of several actions in a sequence, difficult. The service should maintain the state of every client, but UPnP by itself does not provide a way to differentiate clients. Since this is just an example of a limitation, the freedom in defining the API means that the expressibility for service designers is left to the used SDK.

2.7 Alternatives

Although UPnP is evolving rapidly there are few studies available with respect to usability and performance. For the targeted domain the standard is expected to perform reasonably well. However, a wider applicability would be nicer and performance issues become more problematic when the standard increases in acceptance. Therefore, we study alternatives for the points we signaled above and compare them with the standard. This might be used into the evolution of new versions of the standard.

The weakest spot in UPnP from our point of view is the eventing system as it makes data-centered UPnP systems quite difficult to develop. In view of our comments we investigated the following.

- Having subscriptions per event rather than per service;
- allowing events which are not bound to a specific state variable;
- transmitting event-specific data to the subscriber when the event has occurred;
- changing the transport protocol from TCP to UDP in order to avoid TCP timeouts and the necessity to build up and destroy the connection every time the notification has to be delivered. The protocol must be reliable.

According to GENA specification [12] every request should be followed up by a reply. However, in most cases GENA is relying on underlying carrier protocol (HTTP) in these request-reply interactions. Moreover, it suggests that request and reply are transmitted within the same TCP session, and does not provide any facilities to bind a specific request to a correspondent reply. For the UDP-based implementation, the mechanisms for coupling request - reply pairs have to be "weaved" into the message specifications.

3 Eventing Improvements

The first improvement is to decouple events from state variables and to allow to subscribe to particular events rather than to all events of a service. This means that the publisher (device) of the event has to provide the description of the available events to the subscribers (control points). This is done using standard UPnP description mechanisms, and the event is addressed through a URL. The most important additional improvements deal with the transport protocol.

3.1 Protocol Properties

In standard UPnP the GENA protcol is used for passing the events from the service to the subscribed clients, and also for performing the subscription calls from the clients to the service. It uses HTTP as a carrier which, in turn, is based on TCP. However, since we want to use UDP as carrier instead of TCP we need to introduce an adaptation layer which will deal with the differences (Message Delivery Protocol (MDP), see 3). Our protocol is developed with the following properties and design decisions.

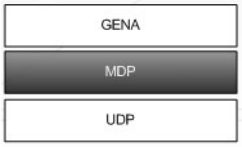

Figure 3. Overview of Eventing Protocol Stack

1. The protocol is GENA-based, i.e., it transports as the smallest unit of information a GENA message, which has a certain format and variable length.
2. The protocol has to deal with the faulty nature of UDP. This means that message losses, duplicated and out-of-order messages have to be handled.
3. The protocol has to couple requests and responses of GENA.

We decompose the eventing protocol into two sub-protocols: the *transport-level protocol* (TMDP), which is in our case UDP-based and independent of the payload, and the *application protocol* (ADMP) which handles all eventing-related information.

3.2 Transport Message Delivery Protocol

TMDP is used for transport GENA messages over the network. Every TMDP message carries one GENA message. The only goal of TMDP, as stated above, is dealing with the faulty nature of UDP. Since UDP is not a reliable protocol, communicating parties have to make sure themselves that no information is lost in the communication. The only way to find out whether a message was received is to obtain feedback from the receiver. The sender can either query the receiver or the receiver can notify upon receipt. The second option is more attractive as it obviously creates less traffic. However, the receiver generally doesn't know when it should receive the message; thus, it cannot inform the sender about failed transmissions, but only about successful ones. Therefore, the sender has to detect failed transmissions by itself. In our case, if within a certain time (we call it `CONFIRMATION_TIMEOUT`) the sender does not receive a confirmation from the receiver, it assumes that the message was lost and it retransmits the message. The number of retransmissions is limited (`RETRANSMISSION_LIMIT`). If the limit of repeated retransmissions is reached, the sender can take action. For example, the sender will not try to send the message any more and the (subscription of the) client will be considered as lost. However, this issue is to be solved not on the transport level.

It is important to note that the confirmation messages do not have to be confirmed. However, they also may be lost. Therefore, we can have situations, when the same message is received more than once. The receiver has to take that into account, and ignore such "double" messages. We use a numbering scheme to discriminate between messages within the same session; the session identification is determined by GENA. Each message that needs confirmation (i.e., all messages except confirmations) has a special field - `MSGID`. This `MSGID` is assigned by the sender, it is unique within

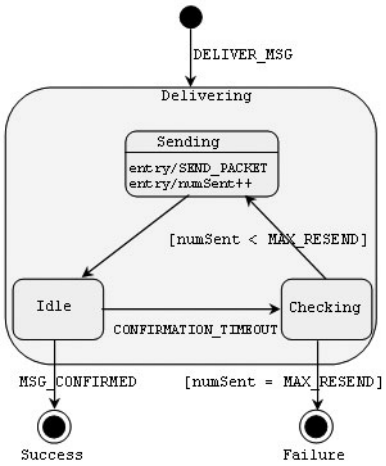

Figure 4. State Machine for TMDP

a session and it will increase in subsequent messages. The receiver extracts this value from the message and puts it into the `IRT` ("In Response To") field of the confirmation message. Every message that has a `MSGID` field has to be confirmed and is sent in a special way. A timer task is created, which performs only one action - transmission of the corresponding datagram. This task is scheduled periodically with a period equal to `CONFIRMATION_TIMEOUT`. The sender uses the `IRT` field in incoming messages to match with one of the IDs of non-confirmed messages. If a match was found, the sender cancels the correspondent timer task. The task is cancelled as well if the related message was unsuccessfully retransmitted for `RETRANSMISSION_LIMIT` times. A state machine for the protocol is presented in Fig. 4.

The `MSGID` field also helps to ignore the messages received more than once. According to our specification the IDs of the messages can only increase. The receiver has to remember the ID of the last received message per sender. The incoming message is functionally ignored unless its ID is larger than this recorded value. It is always confirmed. This checking is performed on a per-subscription basis: if the same control point is subscribed to two different events from the same service the different notifications will not interfere.

3.3 Application Message Delivery Protocol

This part of the protocol deals with the subscriptions and the associated state. While in standard UPnP a subscription is to all information of the service in our version a control point subscribes only to events it is interested in. If needed, the subscription is renewed or cancelled. A state chart for the subscriber is displayed in Fig. 5, and a more explicit state transition table is presented in Table 2. The subscriber sends the subscription request to the publisher. The publisher processes the request and assigns an initial ID to the subscription which is reported back to the subscriber through the HTTP reply. This message is used also as confirmation of the fact

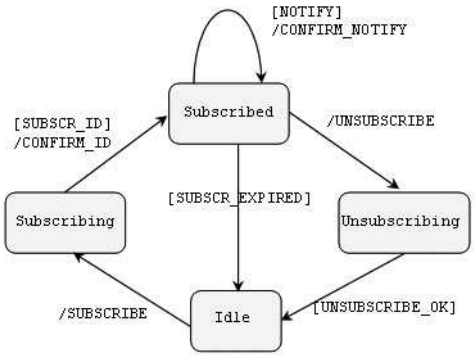

Figure 5. State Machine for Eventing Protocol

that the publisher successfully received the subscription request. The subscription can be terminated in two ways:

- the subscriber can request it;
- the subscription can expire when the subscription timeout has passed.

The state machine at the publisher side is even simpler and has only two states. Transitions between those are described in Table 1.

STATE	CONDITION	ACTION	END STATE
Idle	Received subscription request	Send subscription ID	Idle
	Received confirmation of subscription ID reception	None	Active
Active	Event was fired	Send event notification	Active
	Received subscription cancellation request	Send cancellation confirmation message	Idle

Table 1. Publisher state transitions

4 Results

As mentioned before, a number of issues which were improved in UPnP eventing were of qualitative nature. The impact of these improvements was motivated by the example of the file access service case study (see section 2). However, the changes in the transport protocol have also quantitative aspect. In the experiments we investigated changes in performance of the UPnP eventing system.

Important performance parameters are the *delivery delay* incurred by an event, the number of events that can reasonably be generated by a service (the maximal *notification rate*), and how this notification rate influences the delay.

Experiments were constructed as follows. A publisher exposed to its clients 4 events. The first event was generated with a frequency of 1 kHz, the second with 500 Hz, the third with 250 Hz, and the last with frequency 125 Hz. Notification rates for the regular version of UPnP depend only on the number of clients and the total number of generated events. In the new version the rate is generally lower as the clients are subscribed per event.

The delivery delay was measured as follows. Assuming a symmetric communication channel, we measured it as half the time which passed from the moment when the notification message was sent till the moment when confirmation of this notification was received.

STATE	CONDITION	ACTION	END STATE
Idle	N.A.	Submitted subscription request	Subscribing
Subscribing	Received Subscription ID	Confirm message reception	Subscribed
	Message timeout reached	Resend subscription request	Subscribing
	Received error message in response to subscription request	Confirm message reception	Idle
Subscribed	Received event notification	Confirm message reception	Subscribed
	Subscription timeout was reached	N. A.	Idle
		Request cancellation of subscription	Unsubscribing
Unsubscribing	Received confirmation to cancellation request	N. A.	Idle
	Received error message in response to cancellation request	N. A.	Subscribed

Table 2. Subscriber state transitions

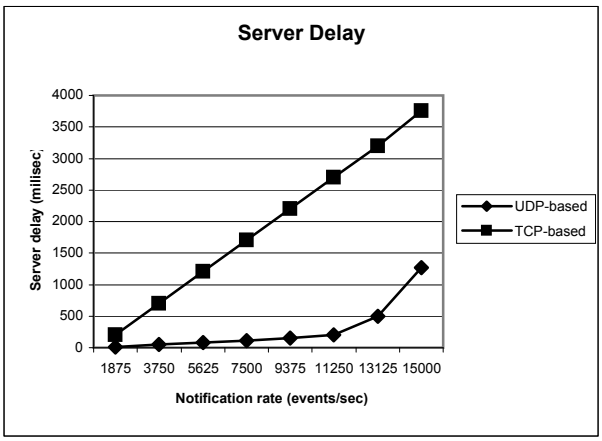

Figure 6. Delivery Delay Comparison

Every client subscribed to all 4 events For every event an event generator was implemented, which raised the event according to its period. After 5000 events the generators were stopped. Number of participating clients per experiment was different and is reflected in notification rate.

For the measurements of the traditional UPnP event delivery delay the set-up was the same. The delivery delay was measured on the service site as a time period from the moment when the notification message delivery started till the moment when the HTTP response to the notification was received from the subscriber. Fig. 6 displays delivery delay as a function of the notification rate. It is clear from the picture that performance of the UDP-based eventing system is significantly better than the TCP-based one shows. Another in-

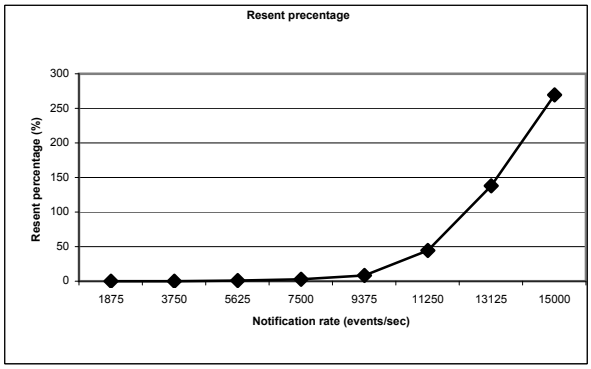

Figure 7. Resent Percentage Dynamics

teresting issue is the type of dependency. For the TCP-based system, it is linear. The situation in the UDP-based one is more interesting. Until a certain point it is linear, and then it becomes exponential. This can be explained by the fact, that message retransmission starts at that point. In order to validate that hypothesis we measured the percentage of the resent messages as a function of the notification rate; this is presented in Fig. 7[1]. Indeed, at the point when the first resent messages appear, the delivery delay grows exponentially.
As we can see the first resent message appears at a quite high notification rate of 5625 notifications per second. According to the measurements, first losses appear at rates around 11000 notifications per second.

5 Future Work

Several interesting issues will be addressed further on. First of all, the protocol can adapt to the changes in the network conditions. For example the `CONFIRMATION TIMEOUT` can be adjusted based on the history of delivery delays. Further, this timeout can be differentiated based on subscriber, so the messages to different subscribers would be retransmitted with a different period.
Prioritization of events would influence the notification delivery times in such a way, that higher priority events are delivered faster than ones with the lower priority.
Often, the subscribers are leaving the network without properly informing the environment about it. Due to that fact, there are often valid subscriptions for invalid subscribers. Publishers can apply various strategies to handle the situations when the messages cannot be delivered to the destination.
With respect to the transport it would be interesting to employ multicasting for event notification, as multicasting corresponds to the nature of this publish-subscribe systems. However, building a reliable and time-efficient protocol based on multicasting is not easily done.

6 Conclusions

In this paper we analyzed UPnP with respect to usability and performance. We altered UPnP specification with the following:

- allowing subscriptions per event rather then per service improves network utilization, flexibility of design and usability of the UPnP services
- decoupling events from the service state variables, adds more flexibility in service design and allows to naturally design data-centered UPnP services;
- transmitting event-specific data along with the event notification enhances the notion of event in UPnP

Further, we changed the transport protocol of the UPnP eventing from TCP to UDP. Unreliability of UDP is addressed by the participating entities - they employ simple retransmission mechanisms in order to improve the chances of the datagram to get to the destination. Experiments show, that retransmission comes to play only when the number of notifications per second exceeds 5000.
The proposed solution perform significantly better than traditional UPnP.

[1] Resent percentage of more than 100% means that some of the messages were resent more than once.

References

[1] J. Webber, S. Parastatidis. Demystifying Service-Oriented Architecture. *Web-Services Journal. Vol. 3, issue 11.*

[2] K. Channabasavaiah, K. Holley, E. M. Tuggle, Jr. Migrating to a Service-Oriented Architecture. *IBM DeveloperWorks*

[3] G. Bieber, J. Carpenter. Introduction to Service-Oriented Programming. *OpenWings White paper.*

[4] UPnP Forum. *http://www.upnp.org* .

[5] J.J.Lukkien, M.F.A. Manders, P.J.F. Peters and L.M.G. Feijs; An Architecture for Web-Enabled Devices. *In proceedings of the 2001 International Conference on Internet Computing, Las Vegas.*

[6] Y. Goland et al. Simple Service Discovery Protocol/1.0. *IETF, Draft draft-cai-ssdp-v1-03, October 28 1999.*

[7] Sergio Marti, Venky Krishnan. Carmen: A Dynamic Service Discovery Architecture. *Mobile and Media Systems Laboratory HP Laboratories Palo Alto HPL-2002-257 September 16th , 2002.*

[8] M. Balazinska, H. Balakrishnan and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. *Pervasive 2002 - International Conference on Pervasive Computing, Zurich, Switzerland, August 2002.*

[9] U. C. Kozat and L. Tassiulas, Network Layer Support for Service Discovery in Mobile Ad Hoc Networks, *in Proceedings of IEEE INFOCOM, 2003.*

[10] C. Bettstetter, and C. Renner. A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. *Proc. 6th EUNICE Open European Summer School: Innovative Internet Applications (EUNICE'00), Twente, Netherlands, September 13-15, 2000.*

[11] T.Tranmanh, L.M.G. Feijs, J.J. Lukkien; Implementation and validation of UPnP for embedded systems in a home environment. *In proceedings of CIIT 2002. St. Thomas, Virgin Islands.*

[12] J. Cohen, S. Aggarwal, Y. Y. Goland; General Event Notification Architecture Base. *http://www.upnp.org/download/draft-cohen-gena-client-01.txt*

[13] A. Rodriguez, J. Gatrell, J. Karas, R. Peschke. TCP/IP Tutorial and Technical Overview. IBM, 2001.