

Improved Eventing Protocol for Universal Plug and Play

Y. Mazuryk, J. J. Lukkien

Department of Mathematics and Computer Science
 Eindhoven University of Technology
 P.O. Box 513, 5600 MB Eindhoven, The Netherlands
 email: {y.mazuryk, 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.

I. INTRODUCTION

Currently an emerging interest to the Service Oriented Architectures (SOA) in the software engineering community is observed. One of the main reasons for such an interest is the amount, diversity and importance of software in the daily life - starting from home appliances and applications, such as TV sets, microwaves, DVD players, etc. and finishing with complex data mining applications, banking solutions, etc. The big role in this process is played by the development of the Internet, and networking in general. Diverse applications, developed by different organizations have to cooperate in order to provide certain end-user functionality. And finding and integrating software in the network context becomes a serious challenge. SOA focuses on solving these problems. It actually stresses interoperability and location transparency.

The basis of SOA is a concept of service. A service is, in fact, functionality provided by a software component, based solely on the interface contract (specification)[3]. Service has a network addressable interface and it can be dynamically discovered and used. Obviously, SOA is a network centered approach, and service-oriented software components can have two major roles: service provider and/or service user. These roles are similar to client and server notions. However, unlike client and server, they do not represent an architectural choice, but roles, that can be played by the same application. Generally, an application in service-oriented context can perform three tasks (see Figure 1):

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;

3. It may use services it finds on the network, either to support the services it exposes or to serve the end-usage;

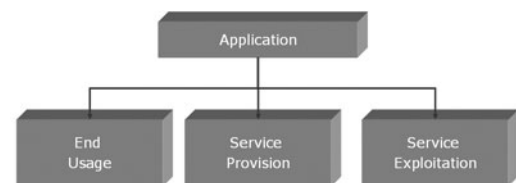


Fig. 1. Tasks of SOA application

Generally, the application can perform at least one of these tasks. Currently, most of applications concentrate on one of the tasks. However, with the advance of SOA, this situation most probably will change, and future applications will be constructed from the services, which could provide end-user functionality as well as employ other network services in order to provide this functionality. Shortly, the structure of the applications will become more complex, and will comprise several levels of service provider - service user interactions. However, the SOA provide means for making such applications less complicated.

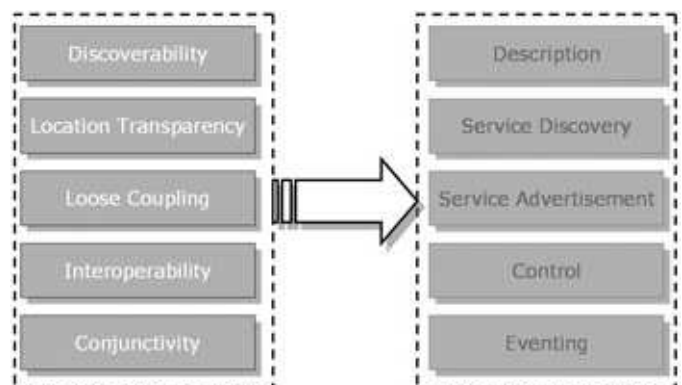


Fig. 2. Properties and mechanisms of SOA

Generalizing this approach to the software development, we conclude that key issues in 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 (see Figure 2) 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.

Currently a number of connectivity standards, which use some ideas of the service-oriented approach, such as JINI [4], JXTA [5] and UPnP [6].

In this article we try to evaluate UPnP as a service-oriented architecture - by analyzing the extend of support of the above mentioned SOA properties in UPnP. We also give some indications about usability and performance of UPnP and propose a number of improvements. Further on we focus on eventing mechanism of UPnP since, in our opinion, it requires more attention then the other four (see Figure 2).

II. UPNP CASE STUDY

The UPnP standard is rapidly growing in popularity, while at the same time the standard is still evolving. We refer to [6] 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.

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 [7], 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.

We decided to see how does UPnP can be used in applications, that it was not designed for. Clearly, data-centered distributed applications were not taken into consideration while designing UPnP. On the other hand, such applications are prevailing in modern networks - including home networking environments.

As the case study, we decided to pick a simple distributed file-sharing application. The architecture of the application is described on Figure 3. There are two roles, which

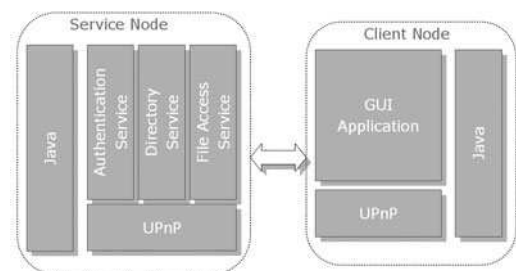


Fig. 3. File sharing application - architecture

can be played by the software components in the application. They can either be a *client node* which provides GUI for access to the shared files, which reside on the service nodes, a *service node* - which exposes services to the network. Service nodes fulfill following requirements: 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;
- service provides certain access control to avoid unauthorized access to its functionality.

In order to satisfy this requirements each service node

comprises three services:

- Authentication service - for access control;
- Directory service - for obtaining information about shared files;
- FileAccess service - for read/write access to the files.

Authentication service employs CERBEROS-like mechanism for access control. Based on the username and password it issues a ticket, which can be used by the user to access directory and file access services. When a ticket is supplied to those two services, they ask authentication component to confirm the validity of the ticket. If the confirmation was successful access to functionality is granted, otherwise - not. Simplified specification of the authentication service is shown on Figure 4. The purpose of the di-

```

ActionList
+ errCode validateUser( [in] username,
                        [in] password,
                        [out] validity )
+ errCode getTicket( [in] username,
                    [in] password,
                    [out] ticket )
    
```

Fig. 4. Simplified specification of authentication service

rectory service is to provide access to the directory of the shared files. That means allowing browsing the directory, obtaining file properties, such as creation date, size, last modified date, etc. The directory service also fires events in case a list of shared files was changed. Simplified specification of the directory service is provided on Figure 5. File service is a rather simple software component which

```

ActionList
+ errCode browseFolder( [in] ticket,
                       [in] folderName,
                       [out] content )
+ errCode getParent( [in] ticket,
                    [in] fileName,
                    [out] folderName )
+ errCode getFileProperty( [in] ticket,
                           [in] fileName,
                           [in] propName,
                           [out] propVal )

StateVariables
+ directoryTree (events when changed)
+ boolean browseChildren
    
```

Fig. 5. Simplified specification of directory service

actually provides opportunities to create and remove files, and to read and write from/to the files.

III. UPnP DISADVANTAGES

A. Discovery

UPnP makes use of Simple Service Discovery Protocol [8] 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 if the network contains a high number of UPnP-enabled devices, multicast traffic will be significant. A number of alternative discovery approaches were introduced (see [9], [10] and [11]). Comparison of various discovery mechanisms is presented in [12].

B. 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. For this, a carrier protocol is needed that is reliable but not connection oriented, since UPnP should support low-latency interaction.

C. 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.

D. 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[13].

The eventing protocol is GENA [14], 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.

E. 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.

F. 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 only 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.

The GENA specification [14] doesn't restrict the request - reply interactions. However, it suggests that request and correspondent reply are transmitted and received within the same TCP session. Indeed, the message formats do not have 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.

IV. EVENTING PROTOCOL

It is essential to decouple event 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.

A. Protocol Properties

In standard UPnP the GENA protocol 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. The reply generated by HTTP is not used. We wish to change as little as possible in GENA. However, since we want to use UDP as carrier instead of TCP we

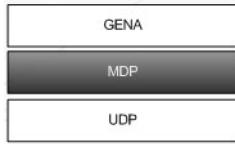


Fig. 6. Overview of Eventing Protocol Stack

need to introduce an adaptation layer which will deal with the differences (Message Delivery Protocol (MDP), see 6). Our protocol is developed with the following properties and design decisions.

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 and duplicated 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.

B. Transport Message Delivery Protocol

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 will not try to send the message any more and the (subscription of the) client will be considered as lost.

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 mes-

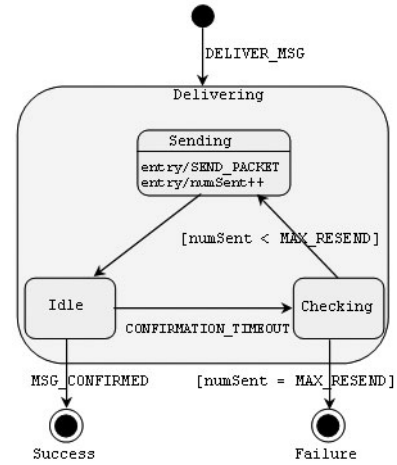


Fig. 7. State Machine for TMDP

sages 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 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 on Fig. 7.

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.

C. Application Message Delivery Protocol

This part of the protocol deals with the subscriptions and the associated state. While in standard UPnP a subscrip-

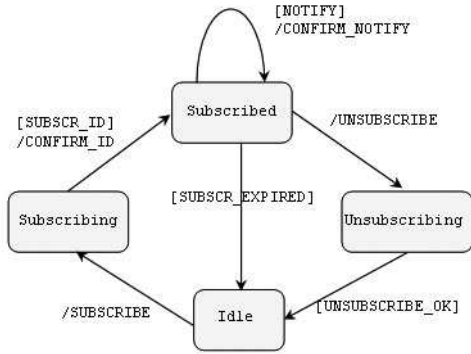


Fig. 8. State Machine for Eventing Protocol

tion 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. 8, and a more explicit state transition table is presented in Table II. 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 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 I.

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 I
PUBLISHER STATE TRANSITIONS

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 II
SUBSCRIBER STATE TRANSITIONS

V. 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.

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.

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. 9 displays delivery de-

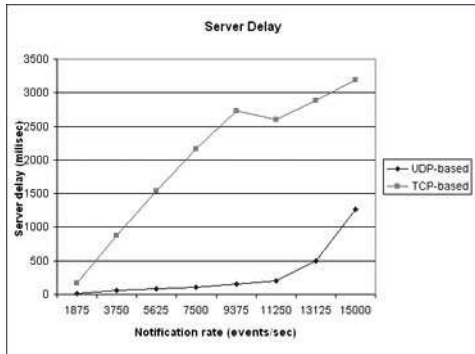


Fig. 9. Delivery Delay Comparison

lay 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 interesting issue is the type of dependency. For the TCP-based system, it is linear, except for a jump at notification rate of 11250 events/sec, which can be explained by congestion control of TCP. The situation in the UDP-based one is more interesting. Until a certain point it is linear, and then it becomes exponential. Theoretically, this can be explained only by the fact that a number messages had to be retransmitted.

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. 11. 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. However, the performance of the protocol can be adjusted by adjusting values of parameters, such as CONFIRMATION TIMEOUT. The delivery delay as a function of confirmation timeout is displayed on 10.

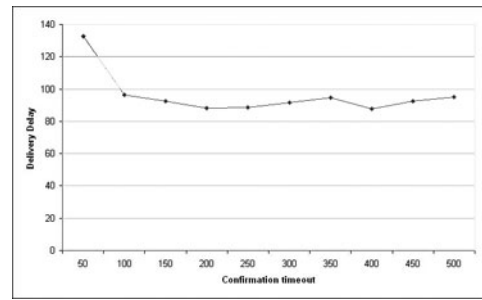


Fig. 10. Delay as a function of retransmission period

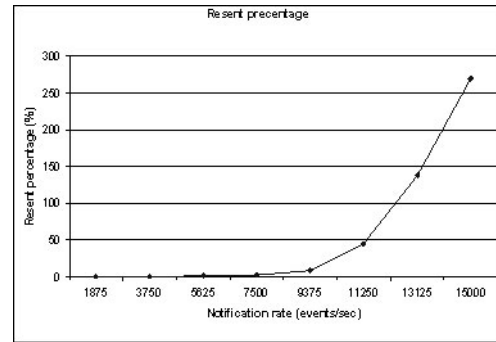


Fig. 11. Resend Percentage Dynamics

VI. CONCLUSIONS AND FUTURE WORK

First of all, changing transport protocol from TCP to UDP brought up some interesting results. Due to the fact, that UDP is connectionless protocol, there is no need to establish and destroy communication session between the publisher and the subscriber each time the message has to be sent from one to another. This results in lower overhead, thus in lower delivery times.

A big number of devices in home networks are mobile; they join and leave the network frequently. Often they do not inform the environment about their intentions to leave the network. That causes troubles on the publisher sides - they try to deliver messages to the non-existent subscribers. Due to the specifics of TCP, it results in waiting for the connection timeouts. Usage of UDP transport completely eliminates this problem, since UDP is connectionless protocol.

The UDP is non-reliable transport protocol, so it does not guarantee message delivery. However, 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. Changes to the architecture of the UPnP eventing add some more design flexibility to it. First of all, the ability to subscribe to a separate event rather to all published events at once is important. For example, an application that is

collecting the history of TV viewing at home would be interested in channel changes, but not in volume corrections. With traditional UPnP eventing it would still receive events about volume changes, and ignore them. But sending out messages which are known to be ignored is a waste of bandwidth. So, per-event subscription would improve the utilization of the network resources.

Tight connection between events and state variables of the UPnP services sometime lead to introduction of artificial state variables, which, in a way "pollutes" service specification. Breaking this connection would give more freedom to a service developer.

Several interesting issues would 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, often there are valid subscriptions for invalid subscribers. Publishers can apply various strategies to handle the situations when the messages cannot be delivered to the destination. For example, if a certain number of messages cannot be delivered to subscriber, the correspondent subscription is automatically cancelled.

With respect to the transport it would be interesting to employ multicasting for event notification. Multicasting is in the nature of publish - subscribe systems. However, it would be a challenge to build a reliable and, in the same time efficient protocol based on multicasting.

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] Jini architecture
- [5] Sun JXTA project homepage. <http://www.jxta.org>
- [6] UPnP Forum. <http://www.upnp.org> .
- [7] 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*.
- [8] Y. Goland et al. Simple Service Discovery Protocol/1.0. *IETF, Draft draft-cai-ssdp-v1-03, October 28 1999*.
- [9] Sergio Marti, Venky Krishnan. Carmen: A Dynamic Service Dis-

- covery Architecture. *Mobile and Media Systems Laboratory HP Laboratories Palo Alto HPL-2002-257 September 16th , 2002*.
- [10] 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*.
- [11] U. C. Kozat and L. Tassiulas, Network Layer Support for Service Discovery in Mobile Ad Hoc Networks, *in Proceedings of IEEE INFOCOM, 2003*.
- [12] 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*.
- [13] 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*.
- [14] J. Cohen, S. Aggarwal, Y. Y. Goland; General Event Notification Architecture Base. <http://www.upnp.org/download/draft-cohen-gena-client-01.txt>