Achieving Eventual Leader Election in WS-Discovery

Filipe Campos
Universidade do Minho
fcampos@di.uminho.pt

José Pereira
Universidade do Minho
jop@di.uminho.pt

Rui Oliveira
Universidade do Minho
rco@di.uminho.pt

***Abstract*—The Devices Profile for Web Services (DPWS) provides the foundation for seamless deployment, autonomous configuration, and joint operation for various computing devices in environments ranging from simple personal multimedia setups and home automation to complex industrial equipment and large data centers. In particular, WS-Discovery provides dynamic rendezvous for clients and services embodied in such devices. Unfortunately, failure detection implicit in this standard is very limited, both by embodying static timing assumptions and by omitting liveness monitoring, leading to undesirable situations in demanding application scenarios. In this paper we identify these undesirable outcomes and propose an extension of WS-Discovery that allows failure detection to achieve eventual leader election, thus preventing them.**

I. DPWS AND WS-DISCOVERY

Th OASIS Devices Profile for Web Services (DPWS) standard [1] specifies a minimal set of requirements that a Web Service implementation must comply with to provide secure messaging, dynamic discovery and description, and simple event notification. Its goal is to bring Service Oriented Computing (SOC) to devices such as small sensors and common appliances, without constraining the behavior and features of richer service implementations on more powerful devices. As protocols and standards that meet these goals for Web Services already exist, this profile defines some constraints and adaptations for their applicability in resource-constrained machines. In particular, DPWS is interesting as the base for self-configurable data centers by building it in management consoles that exist in server hardware.

DPWS builds on standard Web Services concepts of *Client* and *Message* and introduces the definition of a *Device* and *Hosted Service*. A *Device*, also known as *Hosting Service*, handles *Messages* specifically belonging to the standards comprising DPWS, for discovery and description. A *Hosted Service* handles application specific messages and can be addressed directly, i.e., without encapsulation.

A core feature of the DPWS *Hosting Service* is compliance with WS-Discovery [2], which allows services to be located dynamically. Briefly, it assumes that (i) services send an announcement when joining or leaving the network in order to minimize the need for polling and repeated searches; and that (ii) *Clients* can probe for services by type or scope and resolve a service by name. It provides various modes of operation, depending on the availability of a *Discovery Proxy* and to adapt to different scale and resource availability scenarios.

In the Ad-Hoc Mode, entities on the network should make no assumption on the existence of a *Discovery Proxy*. Hence, discovery messages are multicast to the entire network while responses are unicast to the inquiring entity. In Managed Mode, every *Target Service* and *Client* must know the address of a *Discovery Proxy* to enable successful discovery using unicast messages exchanges. Consequently, discovery messages are unicast to a *Discovery Proxy*, which also responds through unicast messages to the enquiring entity.

The way that different entities become aware of the *Discovery Proxy* can be made through different means, such as explicit configuration or even dinamic discovery of the `d:DiscoveryProxy` type itself. This last option paves the way for dynamic mode switching.

The *Discovery Proxy* continuously listens to the well known WS-Discovery multicast group in order to capture any Hello and Bye messages from other *Target Services*, in order to store or update information on them, and Probe and Resolve messages from *Clients* looking for other *Target Services*, to whom the Proxy sends a unicast Hello message.

This configuration option is clearly the most desirable. The use of a *Discovery Proxy* reduces the amount of messages that are multicast, thus reducing network and *Device* resources consumed in scenarios with large number of *Devices*. By dinamically discovering the *Discovery Proxy* itself, it avoids the need for a centralized a priori configuration, that defeats the purpose of WS-Discovery.

Unfortunately, as we describe in Section II coping with failure of Service, and in particular of the *Discovery Proxy*, according to the standard is much less graceful than startup and discovery, as there is no provision for liveness monitoring or operation of multiple Proxies concurrently. We address this in Section III by introducing a small set of changes that allows for eventual leader election, thus providing the key building block on which arbitrary services can provide stability guarantees.

II. PROBLEMS

This simple dynamic discovery protocol has however some shortcomings and can lead to undesirable executions. Consider the following two examples:

1) Newly arrived Proxy: If a new *Discovery Proxy* connects to a network it multicasts a Hello message containing its type. Since *Clients* and *Devices* should consider the leader *Discovery Proxy* to be the one identified in the last discovery Hello, Probe Match or Resolve Match message they received, these entities will then contact this *Discovery Proxy* to fulfill any discovery query. As a *Discovery Proxy* should not perform active searches but update its stored information only by listening to Hello and Bye messages, *Clients* will be querying a Proxy that might have no knowledge of relevant *Devices*.

2) Clients select disparate Proxies: As UDP multicast traffic is not a reliable means of communication, and especially if multiple *Discovery Proxies* boot simultaneously as happens when recovering from a general power failure, *Clients* may end up selecting different *Discovery Proxies*. Note that unless the *Client* repeats its probing step or a Proxy its advertisement (i.e. either of them is restarted), this situation will not be corrected and *Clients* will not be able to make use of common services.

Similar scenarios are possible regarding specific services other than the *Discovery Proxy*, although in that case the potential large scale might call for more elaborate solutions [5]. Moreover, if *Clients* rely on not obtaining different *Devices* for the same discovery probe, the current situation is even more inadequate. In this paper, we restrict our presentation to cases where eventual agreement on the Proxy is sufficient.

III. Leader Election

Our proposal to minimize these problems is to slightly extend WS-Discovery such that the selection of the *Discovery Proxy* when transitioning from Ad-Hoc to Managed Mode leads to eventual leader election. The first challenge is to select a protocol that fits the assumptions of the DPWS. In particular, a protocol that depends on knowledge of number of processes and a higher bound on process faults [3] is not adequate when the system's purpose is discovery and self-configuration. The alternative is a protocol that rests solely on timing assumptions, preferably if it adapts to actual environment [4].

The second challenge is to integrate such protocol into DPWS such that no existing assumptions are invalidated, i.e. that existing *Clients* and *Devices* can continue to be used seamlessly together with novel *Clients* and *Devices*, that obey the new protocol. This can be achieved as follows:

First, we propose the use of an additional UDP multicast channel or group to facilitate communication among *Discovery Proxies*. This group, known as Proxies Multicast Group, allows this communication to be performed out of reach of *Devices* and *Clients*. In detail, when a *Discovery Proxy* responds to the new element with a unicast Hello message, it simultaneously sends a copy of it to the Proxies Multicast Group. This way, other Proxies are able to verify the liveness of other Proxies. By using a separate multicast group, this avoids excessive traffic.

Second, we use the value of the `InstanceId` attribute, enclosed in the `d:AppSequence` element exchanged in discovery messages. The standard recommends that this is the number of times a *Device* boots, or, alternatively and more appealingly, the number of seconds since midnight January 1, 1970, at the time of boot. The described approach assumes this last option for the semantics of the `InstanceId` attribute, as the lower is the value, the older the Proxy is, and therefore it is more likely to possess information on more *Devices* and Services in the network.

Each Proxy can thus observe all Probe messages on the `d:DiscoveryProxy` type, as these are always multicast, and replies, on the new multicast channel. This allows monitoring to be done by estimating time to reply [4]. Moreover, by using the `InstanceId` value, an older Proxy will try to override Probe replies and Hello messages from more recent Proxies, thus maintaining a stable leadership.

IV. Conclusions and Future Work

We have shown that by taking advantage of existing features of WS-Discovery and with a minor extension, it is possible to provide eventual leader election semantics in the selection the *Discovery Proxy*, thus inherently avoiding some problem scenarios. As a future work we aim at extending the guarantees such that discovery provides agreement on *Devices* selected for specific services through a consensus protocol. This should provide many of the advantages of a centralized configuration server for DPWS without the need for centralized management.

Acknowledgments

This work is partially supported by FCT grants SFRH/BD/66242/2009 and PDTC/EIA-EIA/109044/2008.

References

[1] Devices Profile for Web Services (DPWS) 1.1 OASIS Standard. http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html, 01 July 2009.

[2] Web Services Dynamic Discovery (WS-Discovery) 1.1 OASIS Standard. http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html, 01 July 2009.

[3] A. Mostefaoui, M. Raynal, and C. Travers. Crash-resilient time-free eventual leadership. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 208–217, 2004.

[4] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 551–555, 2007.

[5] W. Wogels and C. Ré. WS-Membership - Failure management in a web-services world. In *WWW (Alternate Paper Tracks)*, 2003.